Mechanistic Modeling of Architectural Vulnerability Factor

ARUN ARVIND NAIR, Advanced Micro Devices Inc.
STIJN EYERMAN, Ghent University, Belgium
JIAN CHEN, Intel Corporation
LIZY KURIAN JOHN, University of Texas at Austin
LIEVEN EECKHOUT, Ghent University, Belgium

Reliability to soft errors is a significant design challenge in modern microprocessors owing to an exponential increase in the number of transistors on chip and the reduction in operating voltages with each process generation. Architectural Vulnerability Factor (AVF) modeling using microarchitectural simulators enables architects to make informed performance, power, and reliability tradeoffs. However, such simulators are time-consuming and do not reveal the microarchitectural mechanisms that influence AVF. In this article, we present an accurate first-order mechanistic analytical model to compute AVF, developed using the first principles of an out-of-order superscalar execution. This model provides insight into the fundamental interactions between the workload and microarchitecture that together influence AVF. We use the model to perform design space exploration, parametric sweeps, and workload characterization for AVF.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling Techniques*

General Terms: Design, Modeling, Methodology

Additional Key Words and Phrases: Computer architecture, analytical modeling, mechanistic modeling, Architectural Vulnerability Factor, reliability, soft errors, reliability, availability, servicability (RAS)

Lizy John's research is supported in part by NSF grants 1117895, 1218474, and AMD. Stijn Eyerman is a postdoctoral fellow with the Fund for Scientific Research - Flanders (FWO). Lieven Eeckhout is supported by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement no. 259295. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the funding agencies. The work presented herein is an enhancement of previously published work by the authors [Nair et al. 2012]. This submission covers significant additional material over previous content. Detailed discussions on the assumptions and approximations in the model, and tradeoffs made, along with accompanying data are presented. A completely new section on the impact of compilers, algorithms, implementations, and input data on AVF and SER is also included.
Arun A. Nair and Jian Chen are currently employed at Advanced Micro Devices Inc. and Intel Corporation, respectively. This work was performed when they were at the University of Texas at Austin. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of their employers.
Authors' addresses: Arun A. Nair, Advanced Micro Devices Inc, 1 AMD Place, Sunnyvale, CA 94085; email: nair@utexas.edu; Stijn Eyerman and Lieven Eeckhout, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; email: {stijn.eyerman, leeckhou}@elis.ugent.be; Jian Chen, 3387 NE 12th AVE, Hillsboro, OR, 97124; email: chenjian@utexas.edu; Lizy K. John, University of Texas at Austin, 1 University Station C0803, Austin, TX 78712-0240; email: ljohn@utexas.edu.

DOI: http://dx.doi.org/10.1145/2669364

ACM Reference Format:
Arun Arvind Nair, Stijn Eyerman, Jian Chen, Lizy Kurian John, and Lieven Eeckhout. 2015. Mechanistic modeling of architectural vulnerability factor. ACM Trans. Comput. Syst. 32, 4, Article 11 (January 2015), 32 pages.
DOI: http://dx.doi.org/10.1145/2669364

1. INTRODUCTION

The mitigation of radiation-induced soft errors has emerged as a key design challenge in current and future process technologies as a result of increasing transistor densities and lowering operating voltages [Baumann 2005; Mukherjee et al. 2003; Borkar 2005; Shivakumar et al. 2002]. However, a significant fraction of radiation-induced faults do not affect the correctness of program execution [Wang et al. 2004]: faults may occur in structures that do not contain program state or contain misspeculated or other state that does not affect the correctness of the program. As soft error mitigation mechanisms incur a significant penalty in terms of area, power, performance, and design effort, quantifying this masking effect of program execution using *Architectural Vulnerability Factor* (AVF) modeling enables designers to devise soft error mitigation mechanisms that meet these goals efficiently. AVF captures the probability that a fault in a structure will manifest as an error in the program output, and it can be modeled using statistical fault injection on RTL models later in the design cycle or estimated using *Architecturally Correct Execution* (ACE) analysis [Mukherjee et al. 2003] or SoftArch [Li et al. 2005] on microarchitectural simulators during the early design phase.

Architects use detailed microarchitectural simulations to study the effect of microarchitectural or parametric changes on AVF in order to determine the best tradeoff between Soft Error Rate (SER), performance, power, and area. However, such simulations are very time-consuming when performed over a large number of workloads, for a large number of instructions, and over a large number of microarchitectural configurations and parameters. Such simulations do not reveal the precise microarchitectural mechanisms that influence aggregate metrics such as CPI or branch prediction rates, providing limited insight into the underlying factors influencing AVF. Statistical or machine-learning-based modeling [Fu et al. 2006; Duan et al. 2009; Cho et al. 2007; Walcott et al. 2007] also does not easily quantify the fundamental interactions between the workload and the microarchitecture, making it difficult to derive insight into the factors affecting AVF. Consequently, workload characterization for AVF becomes a significant challenge.

In this work, we develop a modeling methodology to analytically obtain the AVF of a structure using ACE analysis, in the first order, using statistics collected from relatively inexpensive profiling. The central concept behind this methodology is to divide program execution into intervals [Eyerman et al. 2009]. Each interval is a region of execution delimited by miss events that disrupt the dispatch of instructions. We model the occupancy of state that will eventually be committed by the processor in each interval. The occupancy of correct-path state during each interval is averaged, weighted by the time spent in each interval. The AVF of a structure is then estimated by derating this occupancy with the average proportion of un-ACE bits induced by the workload. Additionally, our model is deliberately constructed to capture the interaction of the various miss events and their combined impact on occupancy. This allows us to derive novel quantitative insights into the workload's influence on the AVF of a structure, which may not be obvious from aggregate metrics. We use the same terminology as Eyerman et al. [2009] to refer to such "white box" models as *mechanistic* models, as they are built from first principles, in contrast to "black box" statistical or machine-learning-based analytical models. Figure 1 presents the general overview of our modeling methodology. Workloads are profiled to capture important metrics required by the model, which

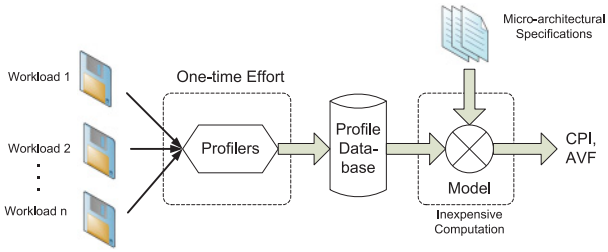


Fig. 1. Design space exploration using the model.

is a one-time effort. Multiple microarchitectures can then be modeled using the data from a single profile. Our methodology can be used to model the AVF of any structure whose AVF correlates with its utilization, and whose utilization is influenced by program execution. In this article, we demonstrate the methodology for estimating the AVF of the reorder buffer (ROB), issue queue (IQ), load and store queues (LQs, SQs), and functional units (FUs). We show that the mean absolute error in estimating AVF for each of these structures is less than 7%, for a four-wide out-of-order machine, as compared to ACE analysis using cycle-accurate simulations.

Our modeling methodology can complement cycle-accurate simulations for performing experiments, such as parametric sweeps or design space exploration. The computational simplicity of the model enables the architect to cheaply explore the design space, eliminate infeasible design points, and guide detailed simulations. Owing to its simplicity, our model also allows the architect to study more workloads over longer runtimes than possible using detailed simulation while providing valuable insight.

The key contributions of this work are as follows:

(1) We present a novel first-order analytical model for AVF, designed from first principles to capture the impact of microarchitectural events on the AVF of major out-of-order processor structures. The key novelty of this modeling effort over prior mechanistic models for performance [Karkhanis and Smith 2004; Eyerman et al. 2009] is that it captures the interaction between different events occurring in a processor, which is required for estimating AVF and estimates AVF with low error. This enables the architect to derive unique insight into the factors affecting the AVF of a structure, not available using aggregate metrics or prior work using black-box models [Fu et al. 2006; Walcott et al. 2007; Duan et al. 2009; Cho et al. 2007].
(2) As the model requires inexpensive profiling, it can be used to perform design space exploration studies nearly instantaneously. In this work, we demonstrate how the model can be used to study the effect of scaling the ROB, issue width, and memory latency on AVF, which provides valuable insight into the effect of microarchitecture and workload interactions on AVF.
(3) We demonstrate how the model can be used for workload characterization for AVF. As the model quantitatively expresses the exact mechanisms influencing AVF, it can be used to identify high or low AVF-inducing workloads. We also demonstrate why aggregate metrics such as IPC or cache miss rates do not correlate with the AVF of a structure.
(4) We demonstrate a use case of the model to study the impact of compiler optimizations, algorithms, implementations, and input data on AVF, SER, and performance to emphasize the importance of workload characterization for AVF.

The remainder of this article is organized as follows: Section 2 revisits AVF modeling using ACE analysis and the interval analysis methodology for modeling performance. We outline our modeling methodology and associated tradeoffs in Section 3. We compare the AVF as predicted by the model to the AVF computed using detailed simulation

in Section 4. Finally, we utilize the model to derive insights into the effect of microarchitectural changes on AVF in Section 5.

2. BACKGROUND

2.1. ACE Analysis

In order to compute AVF, Mukherjee et al. [2003] introduce the concept of *Architecturally Correct Execution* bits. An ACE bit is one whose correctness is required for the correctness of the program. A bit could be either microarchitecturally or architecturally ACE. Bits that are not critical to program correctness are termed *un-ACE*. Microarchitecturally un-ACE bits include bits due to an unused or invalid state, bits discarded as a result of misspeculation, and bits in predictor structures. Architecturally un-ACE bits are a direct result of the instructions in the binary, such as NOPs, software prefetches, predicated false instructions, and dynamically dead instructions.

Mukherjee et al. formally define the AVF of a structure of size N bits as $AVF_{structure} = \frac{1}{N} \times \sum_{i=0}^{N} (\frac{ACE\ cycles\ for\ bit\ i}{Total\ Cycles})$. Thus, AVF of a structure is expressed as the average number of ACE bits per cycle divided by the total number of bits in the structure. AVF is used to derate the circuit-level fault rate of the structure to estimate its *Soft Error Rate* (SER). The derated fault rates for all structures are added up to estimate the overall SER of the processor.

2.2. Interval Analysis

Our first-order mechanistic model for AVF is inspired by earlier work for estimating Cycles Per Instruction (CPI) for out-of-order superscalar architectures, proposed by Karkhanis and Smith [2004] and refined by Eyerman et al. [2009] using *interval analysis*. In the interest of brevity, we only present the basic ideas here. Interval analysis models the program execution as an ideal, miss-free execution, interrupted by miss events that disrupt the dispatch of instructions. Instructions are fetched, decoded, and dispatched into the instruction window or the ROB. Upon completion, instructions are retired from the instruction window in program order. In the absence of any miss events, the processor is able to dispatch instructions at the maximum dispatch rate D[1]. Each miss event interrupts the dispatch of instructions until it resolves. An I-cache miss will cause instruction fetch to be stalled until it has been resolved. A branch mispredict, once detected, will trigger a pipeline flush for all instructions fetched after the mispredicted branch. The branch misprediction penalty for an out-of-order processor is modeled as the sum of the front-end pipeline depth and the branch resolution penalty. The branch resolution penalty is the number of cycles between the mispredicted branch entering the instruction window and the misprediction being detected. A long-latency Last-Level Cache (LLC) miss at the head of the ROB will cause the processor to stall until it is resolved. As an out-of-order processor can extract Memory-Level Parallelism (MLP), and nearly all of the latency of an overlapped miss is hidden behind that of the nonoverlapped data LLC/TLB miss, it is sufficient to count only the nonoverlapped data LLC and TLB miss cycles toward estimating performance for a given instruction window size.

Therefore, the total number of cycles for executing a program is modeled as the sum of the cycles spent in each interval. Miss events that would not interrupt dispatch, such as L1 or LLC data cache hits, are modeled similarly to arithmetic instructions. The model assumes a microarchitecture design with sufficient resources such that the processor would not frequently stall in the absence of miss events while running typical workloads.

[1]Note that D may be less than the peak designed dispatch width if the program lacks sufficient inherent Instruction-Level Parallelism (ILP).

Karkhanis and Smith [2004] and Eyerman et al. [2009] demonstrate that it is sufficient to model these intervals as being independent of one another, with little loss in accuracy. This key simplifying assumption does not hold true for correct-path occupancy. For example, a mispredicted branch that is dependent on an LLC data cache miss significantly reduces the occupancy of correct-path (ACE) bits in the shadow of the LLC miss and is nontrivial to estimate using the existing interval analysis model or aggregate metrics. It is therefore necessary for our AVF model to account for such interactions.

The profilers for the model are implemented as sliding windows and collect statistics for a range of instruction window sizes [Eyerman et al. 2009]. Thus, ROB size, issue width, front-end pipeline depth, and the latencies of instructions, caches, TLBs, and main memory can be changed in the model without requiring additional profiling. If the cache hierarchy or the branch predictor is changed, the corresponding profiler would need to be rerun. Our model retains the same flexibility as the original interval analysis modeling methodology to allow easy design space exploration.

3. MODELING AVF USING INTERVAL ANALYSIS

The unique requirements of our model are to capture the occupancy of state in a given structure in the core, while discarding the occupancy of un-ACE wrong-path instructions, and faithfully modeling the complex interactions between miss events that affect AVF.

In this section, we describe the methodology for estimating the occupancy of correct-path state of the ROB, LQ, SQ, IQ, and FUs, which contain the largest amount of corruptible state in the core (we consider all caches and TLBs to be outside the core). We then estimate AVF by derating this occupancy by the fraction of bits introduced into a structure that were un-ACE. We identify un-ACE instructions through profiling and use this information to determine the number of ACE bits injected into each structure while running the workload. The separation of the program's influence on the number of ACE bits induced in a structure and the residency of these ACE bits in the structure enable the architect to gain deeper insight into the interaction of events and their contribution to overall AVF. As with any analytical modeling methodology, we seek to balance accuracy with simplicity of formulation, the ability to provide quantitative insight, and ease of collecting necessary program characteristics.

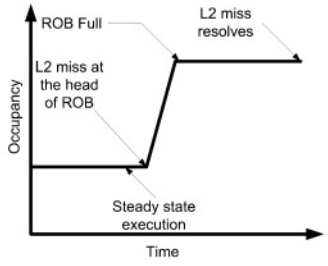
Our modeling methodology must account for the effect of interaction—ignored in interval analysis for CPI—between miss events on the occupancy of correct-path state. The data necessary to achieve this is easily obtained from the profiler for interval analysis, incurring negligible overhead.

We estimate the occupancy of eventually committed state in the ROB in Section 3.1. The ROB occupancy governs the occupancy of LQ, SQ, and FUs. As the IQ[2] can issue instructions out of order, its occupancy is estimated independently, as described in Section 3.2.

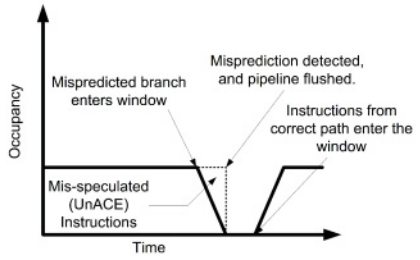
3.1. Modeling the AVF of the ROB

In this section, we study the effect of each miss event on the occupancy of the ROB independently of one another and analyze the impact of interaction between miss events. As illustrated in Figure 2, we linearize the ramp-up and ramp-down curves for occupancy, with slopes equal to the dispatch rate, in the interest of simplicity. It is assumed for the purposes of this work that the designed dispatch and retire widths for the processor are equal.

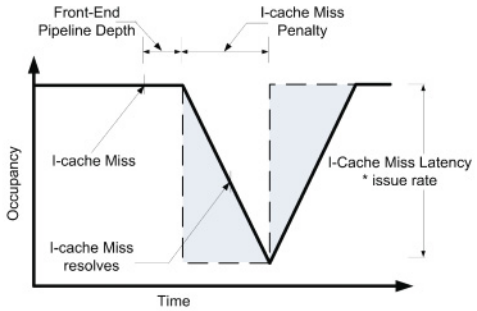
[2]We use IQ or issue queue here to refer to the instruction buffer and scheduler logic. This is the same as the reservation station in a reservation-station-based design. For the purposes of this article, we assume a physical register file (PRF)-based design while calculating the number of bits/entry for the issue queue. Our modeling methodology is equally applicable to either design.



(a) Occupancy of the ROB in the shadow of an L2 miss.



(b) Occupancy of the ROB during a branch misprediction.



(c) Modeling occupancy during an I-cache miss.

Fig. 2. Modeling the occupancy of the ROB using interval analysis.

3.1.1. Occupancy of Correct-Path Instructions. The interval analysis model for performance estimation allows us to estimate the number of cycles spent during each execution interval. Our model allows us to estimate the occupancy of correct-path instructions in the ROB during these intervals. The occupancy of the ROB is modeled as the average occupancy of state during each interval, weighted by the number of cycles spent in each interval, and is expressed as follows:

$$O_{avg}^{ROB} = \frac{1}{C_{total}} \cdot \Big(O_{ideal}^{ROB} \cdot C_{ideal} + O_{DL2Miss}^{ROB} \cdot C_{DL2Miss} + O_{IL1Miss}^{ROB} \cdot C_{IL1Miss} + O_{ITLBMiss}^{ROB} \cdot C_{ITLBMiss} + O_{brMp}^{ROB} \cdot C_{brMp} + O_{DTLBMiss}^{ROB} \cdot C_{DTLBMiss} \Big). \quad (1)$$

In this equation, C refers to the total number of cycles, and O refers to the occupancy of state during each interval. In the following sections, we describe how the occupancy of state during each miss event is computed. C_{total} refers to the total number of cycles required for executing the program/trace.

3.1.2. Modeling Steady-State Occupancy. Given an instruction window of size W, the total number of cycles taken to execute all instructions in the instruction window is a function of the latency of executing the critical path. The critical path, or the critical dependence chain, is the longest-latency dependence chain in the instruction window. The rate at which all instructions in the instruction window can be retired is fundamentally limited by this critical dependence chain. The average critical path length $K(W)$ for a given program is modeled as $K(W) = \frac{1}{\alpha} W^{1/\beta}$ [Michaud et al. 1999; Karkhanis and Smith 2004], where α and β are constants that are determined by fitting the relationship between $K(W)$ and W to a power curve. This analysis is performed

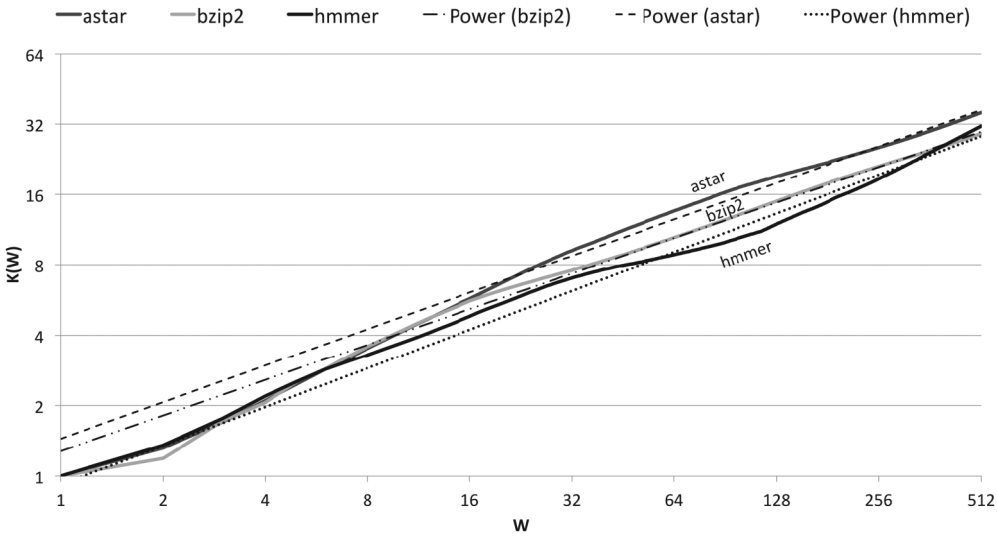


Fig. 3. IW characteristics for sample SPEC CPU2006 workloads plotted on a logarithmic scale.

assuming that all instructions have unit latency. Therefore, given an average instruction latency l, the critical path would require $l \cdot K(W)$ cycles. Using Little's law, the ideal or steady-state IPC ($I(W)$) that can be extracted from the program given an instruction window of size W is presented in Equation (2) [Michaud et al. 1999; Karkhanis and Smith 2004]. For a processor with a designed dispatch width D, setting $I(W) = D$ and rearranging the terms in Equation (2) give us the steady-state ROB occupancy, O_{ideal}^{ROB}, or $W(D)$ necessary to sustain the peak dispatch rate:

$$I(W) = \frac{W}{l \cdot K(W)} = \frac{\alpha}{l}.W^{(1-1/\beta)} \tag{2}$$

$$\therefore O_{ideal}^{ROB} = W(D) = \left(\frac{l \cdot D}{\alpha}\right)^{\frac{\beta}{\beta-1}}. \tag{3}$$

Equation (2) is referred to as the *IW characteristic* [Karkhanis and Smith 2004; Eyerman et al. 2009]. If the steady-state IPC of the program is less than the designed dispatch width for a given instruction window size, the program requires a much larger instruction window to extract the necessary ILP. In this case, the occupancy of the ROB will saturate to 100%. As noted by Karkhanis and Smith [2004], a processor that has a balanced design for a typical workload will not frequently stall due to a full IQ. Therefore, we consider only the typical case for our modeling. An implicit assumption of Equation (2) is that the product of the longest dependence chain and average latency ($l \cdot K(W)$) is approximately equal to the latency of executing the critical dependence path. This approximation may induce errors in pathological workloads with few miss events. However, the separation of critical path and average latency profiling allows us to easily change instruction or cache latencies in the model without reprofiling the workload.

Figure 3 presents the relationship between K(W) and W for a range of instruction window sizes up to 512 entries, plotted on a log-log scale. The power curve fit for determining α and β for each workload is represented by a dashed line. The y-intercept for the fitted IW characteristic is equal to $1/\alpha$ and is a scaling factor. The slope of the line determines $1/\beta$, which represents ILP: workloads with low values of β have lower ILP and, consequently, lower steady-state IPC. Note that $\beta \geq 1$. For the SPEC CPU2006

workloads studied in this work, β ranges between 1.24 and 2.39. This suggests an approximately square-root relationship between inherent ILP and instruction window size, as reported in prior work [Riseman and Foster 1972; Michaud et al. 1999].

We use the coefficient of determination R^2 to estimate the goodness of fit. $R^2 = 1$ indicates a perfect fit, whereas $R^2 = 0$ indicates the absence of a fit. For workloads such as *bzip2*, the fit with the power curve is nearly exact ($R^2 \approx 1$). This fit for practical ROB sizes (between 32 and 512) is apparent from visual inspection of Figure 3. For other workloads such as *astar*, the fit slightly diverges in this range ($R^2 = 0.99$). These two cases are typical of most workloads, and most workloads have R^2 around this mark. However, *hmmer* has $R^2 = 0.98$ due to an irregular IW characteristic that leads to a relatively suboptimal fit as indicated in Figure 3. Although the general trend is approximately that of a power curve, the curve itself has changing behavior in different intervals, leading to divergence with the fitted power curve at various points along it. This may induce errors for workloads that have few miss events, making the accuracy of the IW characteristic fit a critical factor, as will be discussed in Section 4.1.

It is recommended that a value of W that is much larger than the range of instruction window sizes of interest be selected for doing curve fitting. This avoids any error from the divergence at the extremes. In this work, we evaluate the IW characteristic for ROB sizes of up to 512 entries.

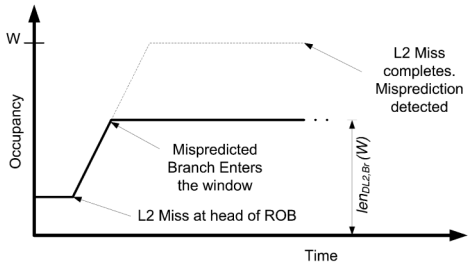
3.1.3. Modeling Occupancy in the Shadow of Long-Latency Data Cache Misses. As shown in Figure 2(a), a nonoverlapped data L2[3] miss (or a TLB miss for a hardware-managed TLB) reaches the head of the ROB, blocking the retirement of subsequent instructions. The processor continues to dispatch instructions until the ROB fills up completely. Thus, the occupancy in the shadow of a nonoverlapped L2 miss is $O^{ROB}_{DL2Miss} = W$. When the data eventually returns from main memory, the L2 miss completes, and the processor is now able to retire instructions. However, the occupancy of the ROB need not return to steady state after the L2 miss completes; it can remain at nearly 100% if the processor is capable of dispatching and retiring instructions at the same rate. In Section 3.1.5, we explain the procedure for accounting for this interaction.

3.1.4. Modeling Occupancy During Front-End Misses.

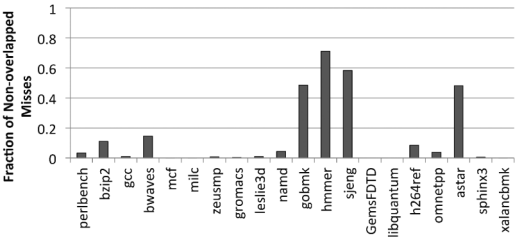
Modeling Occupancy During an L1 I-Cache Miss. The occupancy of the ROB during an L1 I-cache miss depends on the hit latency of the L2 cache, as shown in Figure 2(c), and therefore requires special modeling. When an L1 I-cache miss occurs, the processor is initially able to dispatch instructions until the front-end pipeline drains. Subsequently, the occupancy of the ROB decreases by a rate determined by the ideal IPC (see Equation (2)), as depicted by the solid line in Figure 2(c). Once the I-cache miss resolves and the front-end pipeline is refilled, occupancy of the ROB starts increasing at the rate of the ideal IPC (Equation (2)). Linearizing ramp-up and ramp-down,[4] the shaded areas under the ramp-up and ramp-down are equal, allowing us to approximate occupancy as depicted by the dotted line in Figure 2(c). Thus, the reduction in correct-path state during an I-cache miss is $lat_{L2} \cdot D$, where lat_{L2} cycles is the I-cache miss latency (i.e., the hit latency of the L2 cache), and D is the steady-state dispatch or retirement rate. Thus, $O^{ROB}_{IL1Miss} = O^{ROB}_{ideal} - lat_{L2} \cdot D$. This allows us to model changes in occupancy as steps, greatly simplifying computation, and is used to model other miss events as well.

[3]We assume that the L2 cache is the LLC in the remainder of the study with no loss in generality.

[4]Although it is possible to compute the exact ramp-up and ramp-down curves using Equations (2) and (3), the error due to linearization is negligible.



(a) Mispredicted branch dependent on an L2 miss.



(b) Quantifying the number of non overlapped data L2/TLB misses followed by long intervals, before the occurrence of a front-end miss.

Fig. 4. Modeling the effects of interactions between miss events on occupancy.

The occupancy during other front-end misses such as L2 cache instruction misses and I-TLB misses can be modeled along similar lines. As the latencies of L2 instruction misses and I-TLB misses are relatively large, the occupancy of the ROB goes down to zero.

Modeling Occupancy During a Branch Misprediction. Figure 2(b) illustrates the effect of a branch misprediction on the occupancy of the ROB. The solid line depicts the occupancy of correct-path instructions in the ROB. All instructions fetched after the mispredicted branch are eventually discarded, and hence un-ACE. As correct-path instructions are retired and instructions from the mispredicted path continue to be fetched, the occupancy of ACE state decreases. The overall occupancy, as indicated using the dotted line, remains at the steady-state value, until the branch misprediction is detected and the pipeline is flushed. Karkhanis and Smith [2004] show that assuming an oldest-first issue policy, the mispredicted branch is among the last correct-path instructions to be executed in the instruction window. Simultaneously, retirement of instructions drains the ROB of correct-path state, resulting in low ACE occupancy by the time the misprediction is detected, and the pipeline is flushed. Thus, $O_{brMp}^{ROB} \approx 0$. After the front-end pipeline refills and dispatch resumes, the occupancy of the ROB eventually returns to the steady-state value.

3.1.5. Modeling Interactions Between Miss Events.

Dependent Branch Mispredictions in the Shadow of a Long-Latency Load Miss. Consider the case in which a branch is dependent on a long-latency load miss and occurs within the same instruction window. If such a branch is mispredicted, all instructions in the ROB fetched after the branch instruction are un-ACE. As the branch will not resolve until the cache miss completes, the occupancy of correct-path state in the shadow of this L2 miss is not 100%, as shown in Figure 4(a). Programs such as *perlbench, gcc,*

mcf, and *astar* have a significant number of such interactions. Branch mispredictions that are independent of long-latency data cache misses will resolve quickly such that their interaction has a negligible effect on occupancy.

We capture this interaction by computing the number of instances in which a nonoverlapped data L2 or TLB miss has a dependent mispredicted branch in its instruction window ($N_{dep}(W)$). The average number of instructions between the nonoverlapped miss at the head of the ROB and the *earliest* dependent mispredicted branch ($len_{DL2,Br}(W)$, $len_{DTLB,Br}(W)$) is also captured to estimate the occupancy of correct-path state in the shadow of the L2 miss. Note that the mispredicted branch only needs to be dependent on *any* data L2 or TLB cache miss in the instruction window. This computation can be added to the existing profiler for nonoverlapped data cache misses with little overhead. It does, however, require that information on mispredicted branches from the branch profiler be made available to the nonoverlapped data cache miss profiler.

For a total number of nonoverlapped L2 misses $N_{DL2Misses}(W)$, we express the term $O^{ROB}_{DL2Miss} \cdot C_{DL2Miss}$ in Equation (1) as follows:

$$O^{ROB}_{DL2Miss} \cdot C_{DL2Miss} = (len_{DL2,Br}(W) \cdot N_{dep}(W) + W \cdot (N_{DL2Miss}(W) - N_{dep}(W))) \times lat_{DL2Miss}. \quad (4)$$

Equation (4) expresses the cumulative occupancy of correct-path state in the shadow of a data L2 miss. If all the state in the shadow of a data L2 miss were correct-path instructions, the cumulative occupancy would simply be $(W \cdot N_{DL2Miss}(W) \times lat_{DL2Miss})$, where W is the size of the instruction window. However, if $N_{dep}(W)$ such DL2 misses had $len_{DL2,Br}(W)$ correct-path instructions in their shadow, the cumulative occupancy of instructions in the shadow of such DL2 misses would be $(len_{DL2,Br}(W) \cdot N_{dep}(W) \times lat_{DL2Miss})$. The remaining L2 misses, namely, $(N_{DL2Misses} - N_{dep}(W))$, have an occupancy of W instructions in their shadow. Equation (4) combines the two aforementioned cases to give the cumulative occupancy in the shadow of a data L2 miss.

Interaction of Data and Instruction Cache Misses. Our model is affected by two types of interactions between data cache and instruction cache miss events. The first case occurs when an L2 instruction cache or ITLB miss occurs in the shadow of a nonoverlapped DL2 or DTLB miss, resulting in less than 100% occupancy of the ROB. We find that this case is very rare; only *perlbench* is significantly impacted due to its higher proportion of ITLB misses. We follow a procedure similar to the aforementioned case of dependent branch instructions. L1 I-cache misses in the shadow of a nonoverlapped L2/DTLB miss resolve quickly, and hence their interaction has negligible effect on average occupancy.

A second case occurs when there is an extended period of ideal execution between a long-latency data cache miss and a subsequent front-end miss. As described in Section 3.1.3, we made a simplifying assumption that the occupancy of the ROB returns to steady state relatively quickly after a long-latency data cache miss retires, as a result of subsequent front-end misses or fetch inefficiencies. However, the occupancy of the ROB can stay high if the CPU can dispatch and retire at full dispatch bandwidth. This may affect high IPC workloads that experience few miss events. In order to model this case, we measure the fraction of nonoverlapped DL2 and DTLB misses that are separated from a front-end miss event by *at least* $2W$ instructions. This interval length is chosen to be large enough to eliminate misses that occur in the shadow of the L2 and DTLB miss; that is, it must be greater than W. By picking an interval length of greater than $2W$, we only capture the length of long intervals and filter out the impact of very short intervals on average. Figure 4(b) outlines the average number of nonoverlapped misses that are followed by intervals of greater length than $2W$ instructions for the cache and branch predictor configuration outlined for the *wide-issue machine* in

Table I. Processor Configurations

Parameter	Wide-Issue Machine	Narrow-Issue Machine
ROB	128 entries, 76 bits/entry	64 entries, 76 bits/entry
Issue queue	64 entries, 32 bits/entry	32 entries, 32 bits/entry
LQ	64 entries, 80 bits/entry	32 entries each, 80 bits/entry
SQ	64 entries, 144 bits/entry	32 entries each, 144 bits/entry
Branch predictor	Combined, 4K bimodal, 4K gshare, 4K choice, 4K BTB	Combined, 4K bimodal, 4K gshare, 4K choice, 4K BTB
Front-end pipeline depth	7	5
Fetch/dispatch/issue/ execute/commit	4/4/4/4/4 per cycle	2/2/2/2 per cycle
L1 I-cache	32kB, 4-way set associative	32kB, 4-way set associative
L1 D cache	32kB, 4-way set associative	32kB, 4-way set associative
L2 cache	1MB, 8-way set associative	1MB, 8-way set associative
DL1/L2 latency	2/9 cycles	2/9 cycles
DTLB and ITLB	512 entry, fully associative	512 entry, fully associative
Memory latency	300 cycles	300 cycles
TLB miss latency	75 cycles	75 cycles

Table I. As seen in Figure 4(b), with the exception of *hmmer, gobmk, sjeng*, and *astar*, this situation occurs infrequently. The average length of such sequences for these workloads ranges between 300 and 450 instructions. Thus, the fraction of nonoverlapped L2 misses in Figure 4(b) will experience a subsequent region of ideal execution in which occupancy is W, which is included in our calculations for the model. We perform a similar experiment to capture long intervals between two consecutive data L2/TLB misses but find that they are infrequent and affect only workloads dominated by these misses. In such cases, the contribution of the data L2/TLB miss to overall AVF is so large that the effect of the interval between them is negligible and can be ignored. We nevertheless track these cases as they may need to be accounted for if the workload has few miss events.

Clustered Front-End Misses. With the exception of the L1 I-cache miss, the ROB is completely drained after a front-end miss event. As these misses are independent of one another, their impact on occupancy is separable in the first order. As the ROB is not completely drained in the shadow of an I-cache miss, the lagging I-cache miss in two consecutive I-cache misses will experience a lower occupancy than the leading one. We ignore this interaction due to their relative infrequency and their low impact on average occupancy. We study this assumption in further detail in Section 4.1.

3.2. Modeling of the AVF of the IQ

The occupancy of the IQ requires separate modeling as instructions can issue out of order, in contrast to the ROB, which instructions are dispatched to, and retired from, in program order. We assume an oldest-first issue policy for our model. Prior work by Butler and Patt [1992] shows that scheduling policy has very low impact on IPC. We therefore expect that our methodology can be applied to IQs with other scheduling policies. Occupancy of correct-path instructions during front-end misses is modeled in a manner similar to that of the ROB. Ideal occupancy and occupancy in the shadow of a long-latency data cache or TLB miss are modeled differently, as outlined later.

3.2.1. Steady-State IQ Occupancy. Let $A(W)$ be the average number of instructions in a chain of dependent instructions in the instruction window. $A(W)$ is obtained as a by-product of the critical path profiling necessary to determine $K(W)$. The average latency of each instruction in the IQ is $l \cdot A(W)$. The steady-state arrival rate of instructions

Table II. Configuration of Functional Units for the *Wide-Issue Machine*

Unit	No. of Units	Latency
Int ALU	4	1
Int MUL	1	3
Int DIV	1	20 (blocking)
FP ALU	4	3
FP MUL	2	4
FP DIV/SQRT	1	12/24 (blocking)

in the issue queue is either the steady-state ideal dispatch rate $I(W)$ or the designed dispatch width D, if $I(W) > D$. Using Little's law, $O_{ideal}^{IQ} = l \cdot A(W) \cdot min(D, I(W))$ [Karkhanis and Smith 2007].

3.2.2. Occupancy in the Shadow of a Long-Latency Load Miss. When issue of instructions ceases in the shadow of an L2/TLB load miss, the IQ eventually contains only the instructions dependent on such misses. We measure the average number of instructions dependent on the L2 and DTLB misses in the instruction window to determine the average occupancy during such miss events. This profiling can be added to the existing profiler for determining nonoverlapped data cache misses with no overhead. We also capture the effect of interactions between data L2/TLB misses and front-end misses, similar to the procedure outlined in Section 3.1.5.

3.3. Modeling the AVF of LQ, SQ, and FU

The occupancy of the LQ, SQ, and FUs can be derived from the occupancy of the ROB and the instruction mix (I-mix). Additionally, by classifying un-ACE instructions according to the I-mix, the occupancy of each of these units is derated to estimate AVF.

In this work, we assume that each functional unit is pipelined with buffering equal to the width of the data path and has depth equal to the latency of the instructions that it can handle. For example, the integer multiply unit has a latency of three cycles and is capable of doing 64-bit computation. Thus, the number of vulnerable bits in the integer multiply pipe is 192 bits. The model can easily accommodate other implementations of the functional units. Table II lists the functional unit configuration for the *wide-issue machine*. ACE analysis is performed considering the operand size and logical masking; for example, if a 32-bit instruction is sent to a 64-bit ALU, the upper 32 bits are un-ACE. FU utilization can then be estimated using Little's law, as the latency of each arithmetic instruction and the issue rate are known.

Loads and stores enter the LQ and SQ after they are issued and remain there until they are retired. As we have seen in Section 3.2, the average dispatch-to-issue latency for an instruction is $l \cdot A(W)$ cycles. Thus, the LQ and SQ occupancy can be estimated as the fraction of loads and stores in the ROB, adjusted for the average dispatch-to-issue latency of the loads/stores in the instruction stream. Thus, we compute the occupancy-cycle product in the ideal case for SQ as $(N_{stores}/N_{total}) \cdot O_{ideal}^{ROB} \cdot C_{ideal} - l \cdot A(W) \cdot N_{stores}$, where N_{stores} and N_{total} are the number of stores and total number of instructions, respectively. This assumption may be violated if a majority of the stores are dependent on a load that misses in the L2 cache and occur in the same instruction window as the store. In this case, the store will issue only after the load data returns from memory. This is infrequent in the workloads under study but can be easily detected: the analysis is already performed as described in Section 3.2.2. We would only need to capture the proportion of such stores and adjust the occupancy accordingly. All other occupancy-cycle products from Equation (1) are multiplied by the fraction of stores to estimate O_{total}^{SQ}. We improve the occupancy estimation by capturing the number of loads and stores in the shadow of a nonoverlapped data L2/TLB.

3.4. Summary

This section presents a summary of all the data required for the model. The model requires a critical path and average latency profiler to capture average latency l, α, and β in order to model the steady-state behavior. This profiler also captures the average dependence chain length used for IQ occupancy modeling. We also require a front-end miss profiler for capturing the number of instruction cache misses (L1 and L2) and branch mispredictions. This profiler captures additional information for estimating the branch resolution penalty, which is required for estimating IPC. A memory cache miss profiler for data L2 and TLB misses is required for IPC and occupancy modeling. The profiler requires front-end miss information (obtained using the front-end miss profiler) to model interaction between front-end and back-end events. An ACE analysis profiler is used to tag instructions with ACE information that can then be used to compute the AVF of each unit. Note that most of these profilers do not depend on one another and can be run in parallel.

A single set of profiles can be used to vary the sizes of the ROB, LQ, SQ, IQ, and FUs and the latencies of instructions, I-cache, D-cache, I-TLB, D-TLB, L2, and memory. Thus, a design space evaluation that has two different values for each of the 12 parameters listed previously yields 4,096 unique configurations that can be determined instantaneously from a single set of profiles. As a detailed timing simulation for the same would take hours or days to complete, our methodology provides significant speedup, in addition to insight. Our profilers used in this study are not optimized for speed. Nevertheless, we get a speedup of 18× as compared to detailed simulations over all the studies presented in this article.

4. EVALUATION OF THE MODEL

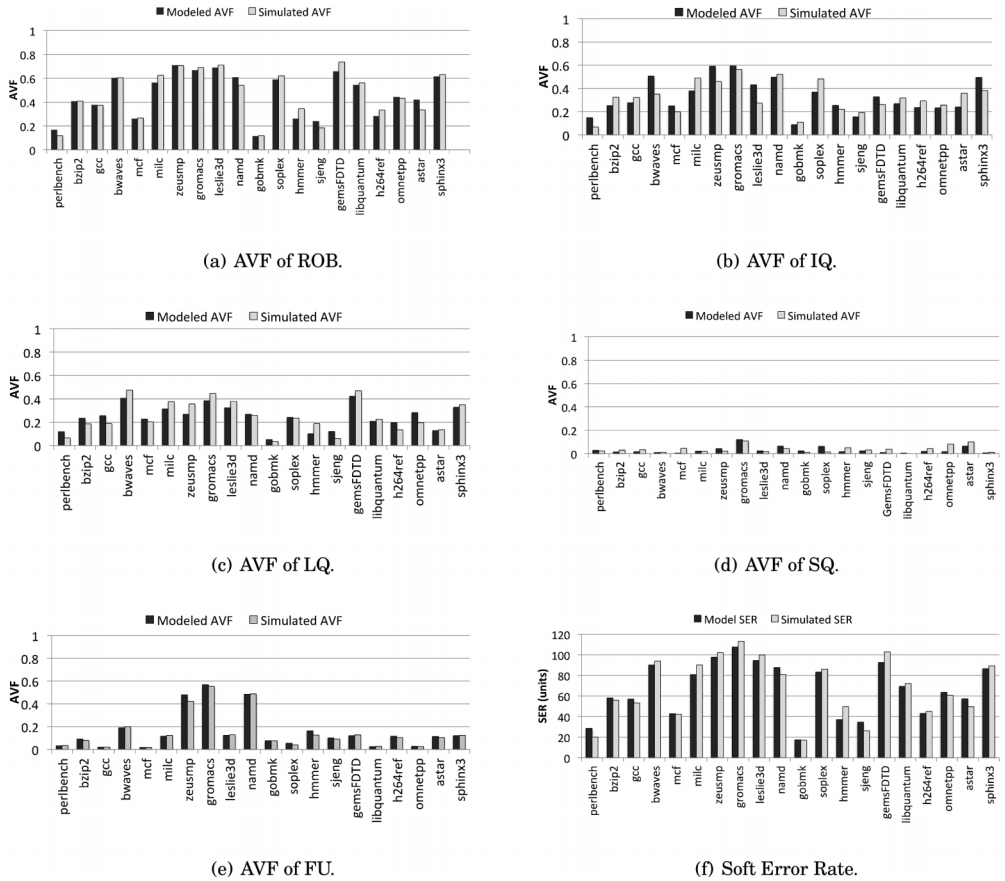
We implement ACE analysis on a modified version of SimpleScalar [Burger and Austin 1997]. We implement detailed, bit-wise ACE analysis in which each entry in the microarchitectural structure of interest has ACE bits fields based on its opcode. For example, stores or branch instructions do not need a result register, and thus the corresponding fields in their ROB entries are un-ACE. We also implement a separate IQ, LQ, and SQ in SimpleScalar. We evaluate the accuracy of our model using 20 SPEC CPU2006 workloads (we were unable to compile the remaining workloads for Alpha) using `gcc` v4.1 compiled with the `-O2` flag. We run the profilers and the detailed simulator on single simulation points of length 100 million instructions, identified using the SimPoint methodology [Sherwood et al. 2002]. The two configurations evaluated in this section are presented in Table I. *Wide-issue machine* and *narrow-issue machine* represent a four-wide and two-wide issue out-of-order superscalar, respectively.

4.1. Results

The AVF of the ROB, IQ, LQ, SQ, and FUs computed using the model and microarchitectural simulation is presented in Figures 5 and 6 for the wide-issue and narrow-issue machine, respectively. We compute the overall SER for these structures assuming an arbitrary intrinsic fault rate[5] of 0.01 units/bit. Each unit expresses the number of failures per unit time. Note that our results are independent of this intrinsic fault rate and would hold true even if a very different fault rate was chosen.

As presented in Table III, the mean absolute error (MAE) and the maximum absolute error are no larger than 0.07 and 0.16, respectively. As AVF is normalized to the number of bits in the structure (Section 2), it amplifies errors in small structures. For a sense of

[5]Fault rates are often measured and expressed in terms of Failure in Time (FIT), which is defined as one failure every 10^9 hours. As real FIT rates are rarely published by manufacturers, we use an arbitrary unit and value instead. Our modeling methodology is independent of the exact FIT rate/bit.



(a) AVF of ROB.

(b) AVF of IQ.

(c) AVF of LQ.

(d) AVF of SQ.

(e) AVF of FU.

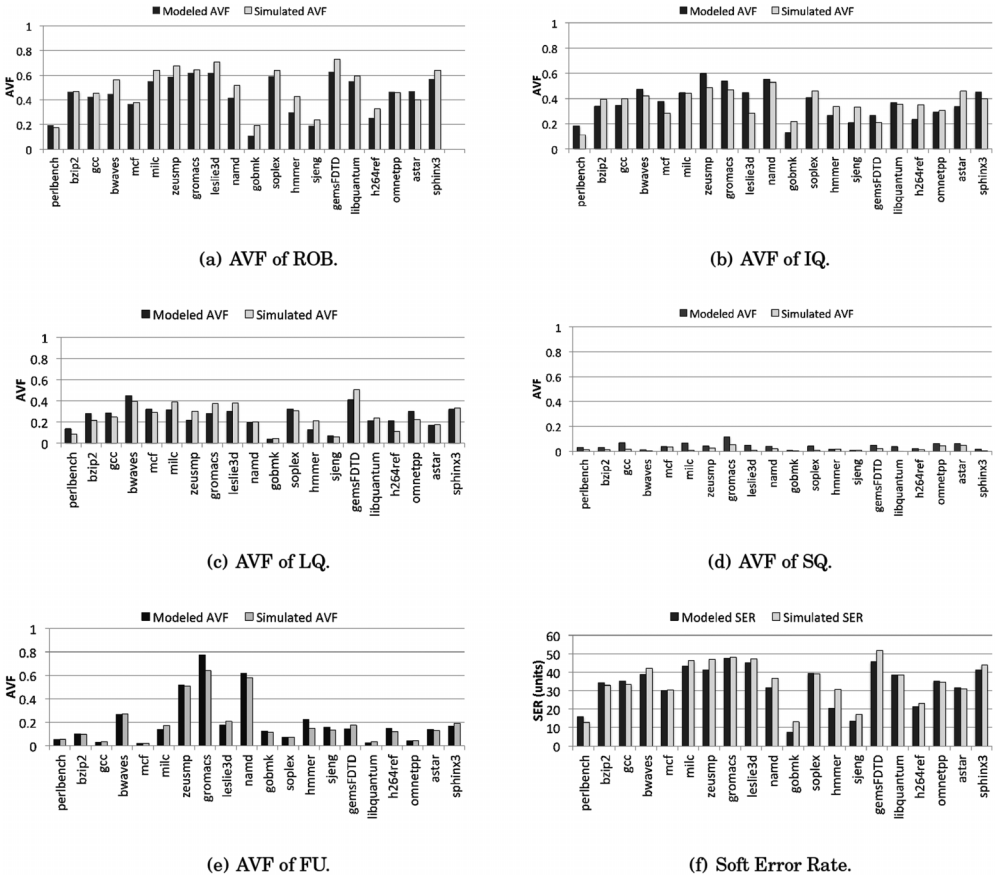
(f) Soft Error Rate.

Fig. 5. Modeled versus simulated AVF for the wide-issue machine.

proportion of the error in computing SER, we express the absolute error in estimating SER in terms of the intrinsic fault rate of each entry in the corresponding structure. In the interest of brevity, in the following discussion, when we express absolute SER error as n entries, we mean "equivalent to the intrinsic fault rate of n entries in the corresponding structure." For example, each of the 128 entries in the ROB has 76 bits per entry and an intrinsic fault rate of 0.01 units/bit. The intrinsic fault rate for the ROB would then be 0.76 units/entry and 97.28 units overall, that is, 97.28 failures per unit time. If the MAE of the model was 3.0 units, this would be equivalent to the intrinsic fault rate of four ROB entries. This gives us an intuitive sense of proportion of the error relative to the intrinsic fault rate of the ROB.

The MAE for estimating SER of the ROB, IQ, LQ, and SQ for the wide-issue machine is 3.8, 4.5, 2.8, and 1.3 entries, respectively. The maximum absolute error for estimating the SER of these structures is 10.2, 9.9, 5.7, and 3.8 entries, respectively. Similarly, the MAE for estimating the SER of the aforementioned structures of the narrow-issue machine is 3.8, 2.1, 1.5, and 0.6 entries, respectively. The maximum absolute error for these structures is 8.3, 5.1, 3.2, and 2.2 entries, respectively.

Figures 5(f) and 6(f) present the combined SER for the ROB, IQ, LQ, SQ, and FUs. Root Mean Square Error (RMSE) is typically used to compute the accuracy of a model and is computed as $\sqrt{\frac{1}{N}\sum_{i=0}^{N}(m_i - a_i)^2}$, where m_i, a_i, and N represent the modeled value,



(a) AVF of ROB.

(b) AVF of IQ.

(c) AVF of LQ.

(d) AVF of SQ.

(e) AVF of FU.

(f) Soft Error Rate.

Fig. 6. Modeled versus simulated AVF for the narrow-issue machine.

Table III. Error in Estimating AVF

	Wide-Issue Machine		Narrow-Issue Machine	
	MAE	Max. Abs. Error	MAE	Max. Abs. Error
ROB	0.03	0.08 *(hmmer)*	0.06	0.13 *(hmmer)*
IQ	0.07	0.16 *(bwaves)*	0.07	0.16 *(leslie3d)*
LQ	0.05	0.09 *(zeusmp)*	0.05	0.10 *(gemsFDTD)*
SQ	0.02	0.06 *(omnetpp)*	0.02	0.07 *(milc)*
FU	0.01	0.05 *(zeusmp)*	0.02	0.13 *(gromacs)*

actual value, and total number of workloads, respectively. RMSE places higher weights on larger deviations, due to the squaring of errors. Normalized RMSE (NRMSE) is computed by dividing the RMSE by the arithmetic mean of the actual values. The NRMSE for our model on the wide-issue and narrow-issue machine is 9.0% and 10.3%, respectively.

The runtime of the profiling is of the same order as functional simulation (or instruction-set simulation), which is typically orders of magnitude faster than detailed execution. Additionally, many of the profiles for a binary are a one-time effort, regardless of the microarchitecture being modeled. Also recall that we collect statistics

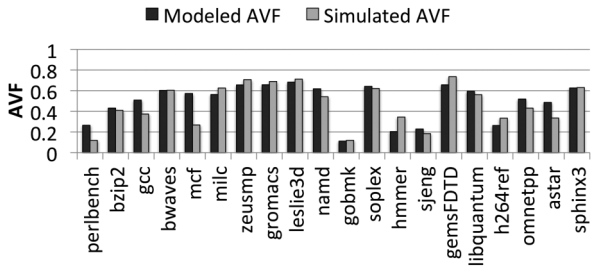


Fig. 7. Impact of ignoring the interaction between miss events on ROB AVF.

over a range of instruction window sizes. Using data from a single set of profiles, we are able to instantaneously vary the sizes of ROB, LQ, SQ, IQ, issue width, instruction latencies, cache and TLB latencies, and memory latency, within typical ranges. Thus, a large design space exploration is possible using the model, with significant speedup over detailed simulation and relatively small error.

4.2. Potential Sources of Error

We outline the various potential sources of error in the model. Awareness of these sources allows the architect to look out for such cases and make the necessary adjustments to avoid them.

—We multiply the proportion of ACE bits injected in a structure by the program with the average occupancy to compute AVF, under the assumption that the proportion of ACE bits induced by the workload remains roughly constant during each interval. We find that this is reasonable over the simulation points used. Over larger execution lengths, a conservative approach would be to estimate AVF over smaller execution lengths and combine the results to determine overall AVF. This does not significantly increase the profiling time or AVF estimation time but may require additional storage.

—For workloads such as *hmmer* that incur very few miss events, the accuracy of the model is strongly dependent on the goodness of fit of the IW characteristic. For workloads with few miss events and for which the relationship between W and $K(W)$ does not exactly fit a power curve, this approximation of fitting the W and $K(W)$ to a power curve may induce errors in modeling the ROB occupancy. In practice, we observe that the error for workloads such as *hmmer*, though higher than most other workloads, is reasonable.

—The out-of-order issue of instructions from the IQ causes errors in the estimation of AVF. For example, NOP instructions, which are un-ACE, may leave the IQ immediately (or not be inserted into it at all) but are included in the computation of $A(W)$ and the average number of ACE bits induced by the instruction stream. Capturing these effects would require the integration of ACE analysis with occupancy profiling. We avoid this approach so that we can gain insight into the architectural and microarchitectural contributors to AVF and avoid rerunning of profiling and ACE analysis on microarchitectural changes such as to the fields in each IQ entry or using a different cache hierarchy.

4.3. Impact of Interaction Between Miss Events

As noted earlier, the interaction between data cache and TLB misses with front-end events affects the accuracy in estimating the AVF of some workloads. Figure 7 illustrates the AVF of the ROB for the wide-issue machine, computed by assuming that miss events are independent of one another. When compared with the ROB AVF error

Table IV. Contribution of I-Cache Misses and Branch Mispredictions in the Shadow of Long-Latency Data Cache Misses to Overall CPI for the Wide-Issue Machine

Workload	Total Performance Penalty Due to I-Cache Misses (%)	Total Performance Penalty Due to Independent Mispredictions in the Shadow of Data Misses (%)
perlbench	6.41	0.33
bzip2	0.00	0.00
gcc	0.01	0.30
bwaves	0.00	2.31
mcf	0.00	0.02
milc	0.00	0.00
zeusmp	0.00	2.04
gromacs	0.00	0.0
leslie3d	0.01	1.03
namd	0.00	1.33
gobmk	6.12	0.67
soplex	0.00	0.78
hmmer	0.00	1.28
sjeng	1.76	0.77
gemsFDTD	0.00	0.1
libquantum	0.00	0.9
h264ref	0.46	0.5
omnetpp	0.37	1.05
astar	0.00	0.0
sphinx3	0.01	1.23

presented in Figure 5(a), it is observed that the impact of such interactions is negligible for a majority of workloads. However, workloads *perlbench*, *mcf*, and, to a lesser extent, *gcc* and *astar* have significant errors. *Mcf* has a significant number of mispredicted branches dependent on data L2 or TLB misses and occurring in their shadow. *Perlbench* experiences instruction TLB misses in the shadow of data L2 or TLB misses, reducing the occupancy of correct-path state. Workloads such as *hmmer* also see an increase in error when the long intervals between a data L2 or DTLB miss and front-end miss are not handled. The mean absolute error in computing the ROB AVF across the entire workload suite in this case is 0.075 (as opposed to an MAE of 0.03 when interactions are considered). A maximum error of 0.3 is observed for *mcf* (as opposed to a maximum error of 0.08 when interactions are considered).

It may be clear from Figure 7 that the AVF induced by *mcf* and *perlbench* would be significantly higher if the front-end miss events occurred outside the shadow of the long-latency data cache miss events. The occurrence of the dependent branch misprediction or I-cache miss relative to the nonoverlapped data cache miss has a negligible impact on CPI.[6] This dependence on the location of miss events relative to one another and to the size of the instruction window further explains why aggregate metrics do not necessarily correlate with AVF, and why AVF is extremely sensitive to the microarchitecture.

It is argued in Section 3.1.5 that consecutive or clustered I-cache misses have an insignificant impact on the estimation of correct-path state due to their low frequency and low latency. Table IV presents the contribution of *all* I-cache misses that hit in the L2 cache toward overall CPI. This is computed as the product of the number of I-cache

[6]As noted in Section 2.2, Karkhanis and Smith [2004] and Eyerman et al. [2009] show that the impact of miss events on CPI is separable.

misses and the I-cache miss latency. In no case do I-cache misses have an influence of more than 6.41% on overall performance, and for most workloads, it is less than 0.5%. Recall from Equation (1) that the occupancy during each event is weighted by its contribution to the total number of execution cycles to determine overall occupancy. To get a sense of the extent of clustering in our workload suite, we measure the proportion of consecutive I-cache miss events that are separated by no more than 50 instructions; as the L2 hit latency is nine cycles on the wide-issue machine, ROB occupancy after an I-cache miss completes will return to steady state in less than 50 instructions. We find that only 25% and 22% of I-cache misses in workloads *perlbench* and *gobmk*, respectively, are clustered, accounting for less than 2% of overall CPI. It is therefore reasonable to ignore the interaction between front-end miss events. It is nevertheless recommended that the I-cache profiler captures the clustering of I-cache misses. If clustering is significant, the impact of the lagging I-cache miss can be modeled in a manner identical to an isolated I-cache miss, but with a lower initial occupancy determined by the leading I-cache miss and the interval length between the two.

Table IV also outlines the contribution of branch mispredictions that are in the shadow of the data L2/TLB miss and independent of long-latency data misses to overall modeled CPI. We capture the number of mispredicted branches in the shadow of the data L2/TLB miss that are independent of the long-latency data miss and multiply it by the average branch misprediction latency. The interval model for CPI does not account for such interactions, as the error in ignoring them is negligible [Karkhanis and Smith 2004; Eyerman et al. 2009]. As argued in Section 3.1.5, it is reasonable to assume that these independent branch mispredictions in the shadow of long-latency cache misses resolve quickly enough such that their overall impact on the average occupancy is negligible, and hence can be ignored. In other words, for the workloads under study, it is reasonable to model these branch mispredictions as if occurring outside the shadow of the blocking long-latency data L2/TLB miss.

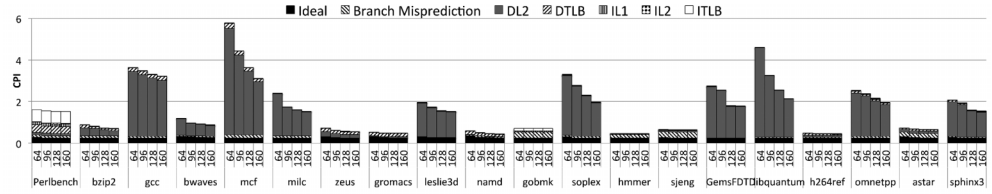
5. APPLICATIONS OF THE MODEL

The analytical model can be used to study performance versus reliability tradeoffs of SER mitigation techniques, the impact of sizing of structures on AVF and performance, compiler optimizations on AVF, different cache sizes and latencies, different branch predictors, and so forth. In this section, we study a small subset of these potential applications of the model. Specifically, we use the model to explore performance and AVF sensitivity to microarchitecture structure sizing, to drive design space exploration, and to characterize workloads for AVF.

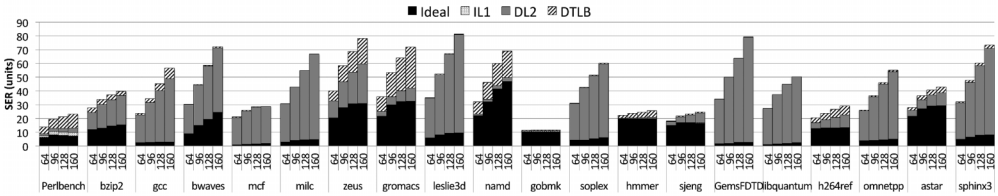
5.1. Impact of Scaling Microarchitectural Parameters

5.1.1. Impact of Scaling the ROB on AVF and Performance. Sizing studies for AVF and performance are interesting because they allow the architect to determine the tradeoff between altering the size of a structure on performance and AVF. For example, it may be reasonable to reduce the ROB size by a small amount provided that it has negligible impact on performance but significantly reduces SER. Using our model, we can instantaneously determine the impact of scaling a structure on AVF and CPI. In this section, we study the impact of sizing the ROB on AVF and CPI on the wide-issue machine presented in Table I, assuming a circuit-level fault rate of 0.01 units/bit. We assume that the IQ, LQ, SQ, and other queue structures are scaled to maintain the same proportion with the ROB as for the wide-issue machine so that the processor is not unbalanced or constrained.

Figure 8 illustrates the impact of scaling the size of the ROB from 64 to 160 entries on the wide-issue machine. The trend in SER due to an increase in ROB size has two general mechanisms. Workloads for which the ROB is not large enough to be able to



(a) Effect of scaling the ROB size on CPI.



(b) Effect of scaling the ROB size on its SER.

Fig. 8. Effect of scaling ROB size on its CPI and SER.

sustain an ideal IPC of four will see an increase in the contribution from ideal execution until this is satisfied. Workloads with MLP will be able to exploit it, resulting in fewer stalls due to data L2/TLB misses. However, for larger ROB sizes, the occupancy of instructions in the shadow of these data L2/TLB misses increases as well, resulting in an overall increase in SER. We present a few examples for, and exceptions to these mechanisms, next.

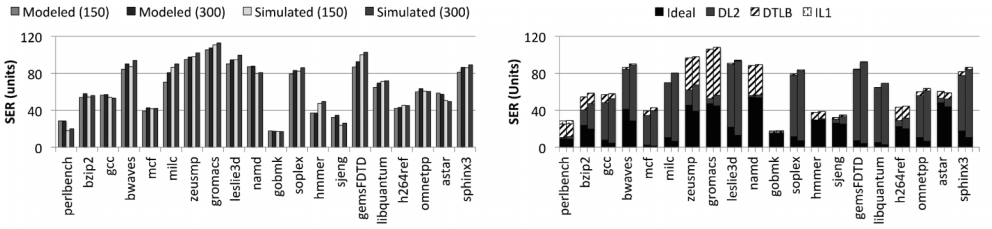
Workloads such as *gobmk* do not see significant change in their CPI or SER due to a large enough ROB and little available MLP. On the other hand, workloads such as *namd* have a long critical dependency path, which results in high values for $\frac{\beta}{\beta-1}$ and l/α (Section 3.1). Consequently, from Equation (3), *namd* induces high SER for all ROB sizes despite its low CPI.

For workloads such as *libquantum*, the increase in ROB size provides increased MLP, resulting in lower CPI, but also a greater SER in the shadow of the L2/DTLB miss. *Libquantum* is able to exploit more MLP than *gemsFDTD,* resulting in a greater rate of reduction of CPI and a lesser rate of increase of SER. *Bwaves* and *zeus* experience an increase in SER due to both mechanisms.

Perlbench and *mcf* represent two important exceptions to this general trend. Despite both workloads having a significant number of data L2/TLB misses, *mcf* experiences a significant number of branch mispredictions dependent on data L2 misses, and *perlbench* experiences I-TLB misses in the shadow of data L2/TLB misses. Consequently, scaling of the ROB size has little impact on SER, as the occupancy of state per cycle does not change significantly. *Mcf* experiences reduction in CPI due to MLP, but the occupancy of ACE state during such misses is still limited by the dependent mispredicted branches.

The scaling study allows the architect to make the appropriate tradeoffs between performance and SER and understand the factors affecting the scaling of workloads. For example, the 128-entry ROB provides a speedup of 1.098 (harmonic mean) and increases the average SER by 18% over the 96-entry ROB.

Figure 8 also provides visual confirmation of the fact that aggregate performance metrics such as IPC, L2 misses, branch mispredictions, and so forth do not correlate well with AVF. From Figure 8(a), it is clear that *gemsFDTD* has fewer stalls due to L2 cache misses when compared to *mcf*. However, it induces much higher AVF in the ROB and overall SER than *mcf*, for reasons noted earlier. *Perlbench* has higher DTLB



(a) Sensitivity of SER to memory latency.

(b) Impact of scaling memory latency from 150 cycles (left) to 300 cycles (right).

Fig. 9. Sensitivity to memory latency.

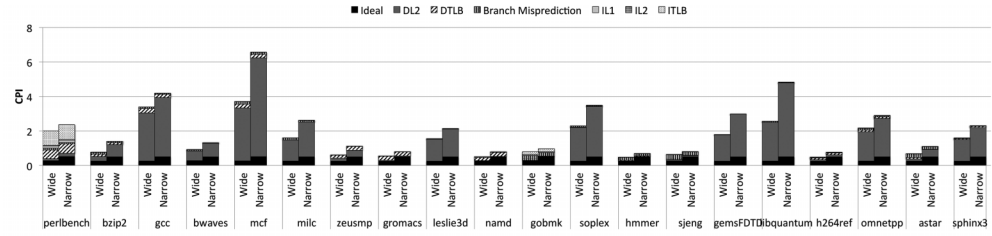
misses than most other workloads and yet induces very low AVF. *Gromacs* is a low CPI workload with significant performance loss due to branch mispredicts and yet induces higher AVF than *h264ref*.

5.1.2. Sensititivity of AVF to Memory Latency. We study the impact on AVF of changing memory latency to provide insight into the influence of memory bandwidth contention in multicore processors, or Dynamic Voltage and Frequency Scaling (DVFS). DVFS can change the speed of the memory relative to the CPU's frequency, making memory appear slower or faster in terms of CPU cycles. Figure 9 presents the overall SER for a memory latency of 150 cycles and 300 cycles obtained using our model and from detailed simulation, assuming a constant circuit-level fault rate[7] of 0.01 units/bit. The memory latency used by the model and simulation is enclosed in parentheses. From the formula for AVF (Section 2), we observe that reduction in memory latency reduces the total number of cycles of ACE bit residency and the total number of execution cycles. Consequently, the change in AVF in Figure 9 is sublinear and thus less sensitive to memory latency when compared with CPI. AVF typically decreases with a decrease in memory latency, although it may increase, as seen with *astar*. In the case of *astar*, the occupancy of state during steady-state execution is high and is a dominant contributor to overall AVF and CPI, as seen in Figures 8 and 9(b). However, nearly 50% of data L2 misses have dependent mispredicted branches in their shadow, resulting in low average occupancy of ACE state. As total SER is the average occupancy of state weighted by the cycles spent in each interval, reduction in memory latency results in a slight increase in overall SER for *astar*.

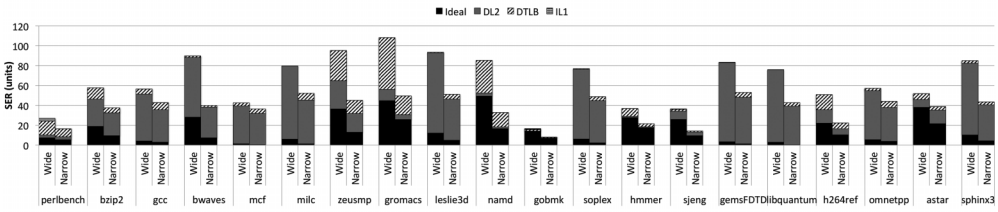
Workload *namd* has negligible data L2 cache misses (0.13 MPKI) on the *wide-issue* machine. Consequently, reducing DRAM memory latency will have little impact on AVF. As noted earlier, the high AVF of *namd* is due to its high values of l/α and $\frac{\beta}{\beta-1}$. As α and β are binary specific and cannot be changed without recompiling or re-engineering the workload, reducing the average latency l is the only microarchitectural knob available to reduce the observed AVF without hurting performance. As *namd* has 23% load instructions and 43% FP instructions, reducing the latencies of these instructions will clearly reduce the AVF induced by *namd*. For example, reducing the L1 data cache load-to-use delay and the FP multiplier latency by one cycle each results in unlocking sufficient issue bandwidth, such that the steady-state occupancy is reduced.

The comparison with simulation also serves to validate our model. The average change in AVF as predicted by the model is 3.25 units, as compared to 2.22 units from simulation. The model faithfully captures the trend for change in SER. Figure 9(b) illustrates the fraction of SER attributable to each event for a memory latency of 150

[7]Although the circuit-level fault rate will significantly increase at low voltages, a constant value allows us to highlight the change due to AVF.



(a) CPI stacks for the wide-and narrow-issue machine.



(b) SER contribution of microarchitectural events for the wide-and narrow-issue machine.

Fig. 10. Comparison of CPI and SER of the wide- and narrow-issue machines.

and 300 cycles. Although the overall AVF remains nearly the same, the contribution of AVF in the shadow of an L2 miss reduces significantly for workloads that are dominated by L2 cache misses. Conversely, the relative contribution from ideal execution and DTLB miss increases. The workload completes faster due to lower memory latency and, consequently, the contribution of steady-state and DTLB misses to the overall occupancy increases.

5.2. Design Space Exploration

The model can be used to compare different microarchitectures for their impact on performance and AVF/SER. Figure 10 presents the CPI and SER of the wide-issue and narrow-issue machines outlined in Table I. The SER is computed for the ROB, LQ, SQ, IQ, and FU and is broken down into its contributing events so as to provide better insight. On average, there is an 81% increase in SER and an average speedup of 1.35 (harmonic mean) going from the narrow-issue to the wide-issue configuration. This is attributable to an increase in ROB size and dispatch width. Unlike scaling the ROB size (Section 5.1), increasing the dispatch width typically increases the SER across all workloads. From Equation (3), a larger instruction window is required to sustain a larger dispatch width. For our workloads, β is between 1.24 and 2.39, resulting in a superlinear increase in the ideal occupancy. Although branch resolution time increases with dispatch width [Karkhanis and Smith 2004; Eyerman et al. 2006], it is reasonable to expect that SER would generally increase with dispatch width on a balanced design. As noted in Section 5.1, *namd* and *bwaves* have a long critical path $K(W)$. These workloads have sufficient ILP for the narrow-issue machine, but not the wide-issue machine, resulting in maximum occupancy of state during ideal execution for the wide-issue case. Additionally, *bwaves* also experiences an increase in SER due to data L2 misses. The SER for *bwaves* and *namd* increases by a factor of 2.26 and 2.6, respectively. On the other hand, *mcf* is unaffected by an increase in issue width or ROB size due to the large number of dependent mispredicted branches in the shadow of its data L2 misses.

5.2.1. Area Versus SER for the Wide- and Narrow-Issue Machine. To understand the implications on multicore design, we compare the SER for a homogeneous Chip Multiprocessor

(CMP) using multiple wide-issue and narrow-issue machines under the same area budget. Using the McPAT simulator [Li et al. 2009], we estimate that on a 32nm process, the wide-issue machine (core+cache) has a 65% higher area than the narrow-issue machine. Given that the wide-issue machine has on average 81% higher SER for the ROB, LQ, SQ, and IQ, a wide-issue multicore CMP would be, on average, more vulnerable for these structures for the same area. Of course, we have not modeled all structures or considered the impact of shared resources in the memory hierarchy, such as memory bandwidth, of the CMPs. However, as the ROB occupancy governs occupancy of state in most other structures in the core, this result gives some insight on the SER of core structures in the CMPs for the configurations under consideration. Although there is a superlinear increase in core SER going from a narrow-issue to a wide-issue machine in general, it should be noted that the exact scaling factor depends on the sizes of the resources in the *wide-issue* versus the *narrow-issue* machine.

5.2.2. The Efficacy of Opportunistic Redundant Multithreading. The model can also be used to provide insight into the efficacy of soft error mitigation schemes. Gomaa and Vijaykumar [2005] propose an opportunistic mechanism, called Partial Explicit Redundancy (PER), of enabling Redundant Multithreading (RMT) [Reinhardt and Mukherjee 2000] during low IPC events, such as L2/TLB miss, and disabling it during high-IPC intervals to minimize the performance loss. RMT employs a lagging thread that reexecutes the operations of the leading thread and detects faults by comparing the output. Load values and branch outcomes are forwarded by the leading thread so that the lagging thread does not incur any miss penalties and always runs in the ideal mode. Using detailed simulation for a specific microarchitecture running SPEC CPU2000 workloads, Sridharan et al. [2007] investigate the efficacy of PER for the ROB, LQ, SQ, and IQ and report that nearly 60% of vulnerability occurs in the shadow of a long-stall instruction, most of which are data L2 cache misses.

Under an optimistic assumption of no performance loss using the opportunistic scheme, the components of SER in Figure 10(b) corresponding to data L2 and TLB misses would disappear. Whereas this scheme generally reduces the AVF of most workloads significantly (we compute an SER reduction of 66% for the wide-issue machine), *namd* would still have high AVF. Furthermore, given its low CPI (Figure 10(a)), the performance of *namd* will be significantly impacted if RMT is enabled. Of course, these results are microarchitecture and workload specific. For example, we compute that PER results in an average SER reduction of 60% when the memory latency of the wide-issue machine is reduced to 150 cycles, as illustrated in Figure 9(b). Our results are similar to earlier work. The model enables architects to estimate the efficacy of such a scheme in the first order for their microarchitecture and workloads.

5.3. Workload Characterization for AVF: A Case Study Using Sorting Algorithms

It is difficult to draw inferences on the effect of a workload on the AVF of a structure using aggregate metrics beyond a qualitative analysis. Aggregate metrics such as cache miss rates or branch misprediction rates (which correlate well with CPI) provide hints, but as noted in earlier sections, there may be exceptions to our general intuitions of occupancy of state. This poses a significant challenge for workload characterization and developing a benchmarking suite for AVF. Nair et al. [2010] demonstrate that SER coverage of a workload suite changes significantly when the microarchitecture or the intrinsic fault rates of structures are changed; a workload suite that may have good coverage on one microarchitecture or intrinsic fault rate may not necessarily have the same coverage on another. As noted in preceding sections, this may be true even when the workload suite has sufficient variety in terms of aggregate metrics used for workload characterization for performance. For example, as noted in Section 5.2,

the AVF induced by *mcf* is low because of the location of the dependent mispredicted branches relative to the L2 cache miss, a characteristic that has low impact on performance. Moreover, workloads may vary in the proportion of ACE bits despite having similar performance characteristics. Observing the change in relative SER going from the narrow- to wide-issue machine may make it apparent that the microarchitecture affects the AVF of different workloads differently, and the mechanisms influencing AVF are not discernable from aggregate metrics used for workload characterization for performance. Our work presents a framework for performing workload characterization for AVF/SER using relatively easy-to-measure metrics that allow the architect to test the coverage of the workload suite.

The model enables the architect to study a greater number of workloads and over longer intervals of execution than may be feasible using detailed simulation, and within the bounds of error of the model, to identify workloads or phases in the workload that induce high AVF in particular structures, enabling better workload characterization for AVF. In the following section, we use the model to characterize the impact of compiler optimizations and algorithms on AVF.

5.3.1. Use Case Experimental Setup. This section presents yet another potential use case for the model for workload characterization. We examine the impact of compiler transformations, algorithms, implementations, and input data on AVF and SER for a given microarchitecture and intrinsic fault rate. An exhaustive study of these factors is beyond the scope of this article; this section is intended to serve as an illustrative use case for the model. We expect that compiler writers, code optimizers, and software developers may use the model to quickly evaluate the impact of their optimizations on SER and AVF and choose the appropriate algorithm or optimization that minimizes these factors while maximizing performance.

The impact of various compiler transformations on AVF and SER has been studied in prior art. However, we are unaware of any work that provides insight into the specific mechanisms affecting AVF. Sridharan and Kaeli [2009] study the impact of compiler transformations on the register file AVF but do not find a specific pattern or identify specific mechanisms affecting the AVF. More recently, Demertzi et al. [2012] study the effect of various compiler flags on AVF in the core but do not explain why a workload compiled with different optimization levels induces different AVFs beyond qualitative observations. They observe that highly optimized code induces higher AVF than unoptimized, or less optimized, code. In the following discussion, we will characterize the fundamental causes of these variations.

We select sorting as a workload to evaluate the impact of the aforementioned issues. Sorting algorithms are widely understood and are frequently used in real programs. Such microbenchmarks allow us to easily map our observations to our intuitive understanding of workload behavior, thereby enabling better insight. Sorting also stresses the memory hierarchy and CPU (and I/O) and is therefore used for benchmarking systems. For example, JouleSort [Rivoire et al. 2007] has been proposed as an energy efficiency benchmark to evaluate both power consumption and performance of a system. We use three *divide-and-conquer* sorting algorithms: *quick sort, heap sort*, and *merge sort*. Each algorithm has time complexity $O(n \cdot lg(n))$. Additionally, we compare an optimized implementation of the quick sort algorithm with a "textbook" recursive implementation to understand the impact of implementation-level optimizations.

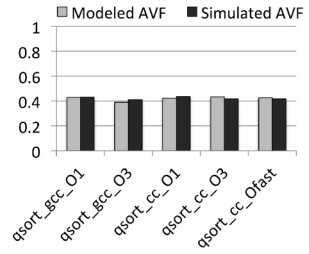
To study the effect of compiler optimizations, we compile the workloads using `gcc` version 4.1, `-O1` and `-O3` optimization flags, and using Alpha's `cc` compiler using `-O1`, `-O3`, and `-Ofast` optimization flags. Optimization level `-O1` enables optimizations that do not significantly increase the code size. Inlining of small functions and dead code elimination are some of the optimizations enabled. `-O3` enables aggressive

optimizations that may involve increasing the size of the code in order to gain performance. Optimizations such as aligning branch targets, loop unrolling, aggressive instruction scheduling, and so forth are turned on. Option `-Ofast` enables even more aggressive optimizations such as fast math. A reliability architect would have to consider all possible compilation options and design hardware to account for the observable worst-case scenario. A compiler writer or application designer would be interested in an optimization that minimizes vulnerability while simultaneously improving performance.

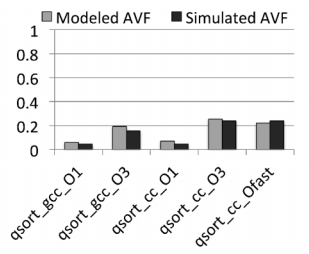
We use the *qsort* benchmark in the *MiBench* [Guthaus et al. 2001] workload suite for our study. This benchmark sorts points in a three-dimensional space based on their distance from the origin. Instead of using a precompiled *glibc* implementation of quick sort, we link our own implementation of the aforementioned sorting algorithms, compiled using the appropriate flags. We create an input file with 500,000 randomly generated x, y, and z coordinates. As the Cartesian distance is computed as a double-precision floating-point value, the data footprint is large enough such that it does not fit in the L2 cache of the *wide-issue* machine. As aggressive optimization often results in a reduction in the number of instructions, simulating a fixed number of instructions may be misleading. Therefore, we demarcate the beginning and end of the call to the qsort routine and simulate/profile all the instructions in between. In the following sections, we study the impact of compilers, algorithms, implementations, and input data on the AVF and SER of the *wide-issue machine*.

We define a new metric to compare the failure rates experienced by a workload while running on a microarchitecture, called *Cumulative Failures (CFs)*. CF captures the average or expected number of failures encountered by a workload running a microarchitecture and is defined as $CF = CPI \cdot f \cdot Icount \cdot SER = (execution\,time \times SER)$, where f stands for frequency. CF is along the lines of other metrics proposed to account for workload and microarchitecture performance while estimating the impact of a microarchitectural change on SER [Weaver et al. 2004; Demertzi et al. 2012]. For example, Weaver et al. [2004] propose *Mean Instructions to Failure (MITF)*, which is defined as $MITF = IPC \cdot f \cdot MTTF$, where $MTTF$ is the Mean Time to Failure of the system. As $MTTF = \frac{1}{SER}$, it is clear that $CF = (\frac{1}{MITF} \cdot Icount)$. CF allows us to compare the failure rates of two workloads running on a given microarchitecture. MITF is useful for evaluating IPC versus SER tradeoffs at a microarchitectural level. Unlike MITF, CF accounts for a change in the runtime of a workload. In this article, we normalize CF to the core frequency for convenience, as the microarchitecture being compared is the same and the frequency is also unchanged.

5.3.2. Impact of Compilers on AVF. In this section, we examine the impact of compiling the recursive *quick sort* algorithm using `gcc` with `-O1` and `-O3` optimization levels, and Alpha `cc` with `-O1`, `-O3`, and `-Ofast` optimization levels. The quick sort algorithm is implemented as described by Cormen et al. [2001]. The first element in the array is picked as the pivot around which the sorting is performed. In the interest of brevity, we only discuss the results for the ROB and the LQ. Figure 11 illustrates the CPI, ROB AVF, and LQ AVF induced by running these binaries on the *wide-issue machine*. Figure 11(c) outlines some important metrics obtained from our profilers. Column N_{Data} lists the number of *nonoverlapped* data L2 and TLB Misses Per Kilo Instruction (MPKI). Column N_{ld} lists the total number of loads Per Kilo Instruction (PKI). Column *I-count* contains the total number of dynamic instructions for sorting the 500,000 points, normalized to the I-count of *qsort_gcc_O1*. The column *Exec. time* reports the execution time for the application in cycles, computed as a product of *CPI* and *I-count*. It may be clear from Figures 11(a) and 11(b) that the binaries induce similar AVF in the ROB, but very different AVF in the LQ.



(a) Impact of compiler optimizations on ROB AVF



(b) Impact of compiler optimizations on LQ AVF

	α	β	O_{ROB}^{ideal}	N_{Data} MPKI	N_{ld} PKI	CPI	I-count (relative)	SER (relative)	Exec. Time (cycles)	CF/GHz
qsort_gcc_O1	1.37	1.34	128	0.60	171.8	0.48	1.0	1.0	0.48	0.48
qsort_gcc_O3	1.58	1.58	32.67	1.10	198.5	0.60	0.54	1.22	0.32	0.40
qsort_cc_O1	1.41	1.33	128	0.60	175.7	0.48	1.0	0.99	0.48	0.48
qsort_cc_O3	1.21	2.26	22.73	1.30	351.7	0.71	0.46	1.34	0.37	0.44
qsort_cc_Ofast	1.21	2.26	22.73	1.30	351.7	0.71	0.46	1.33	0.37	0.43

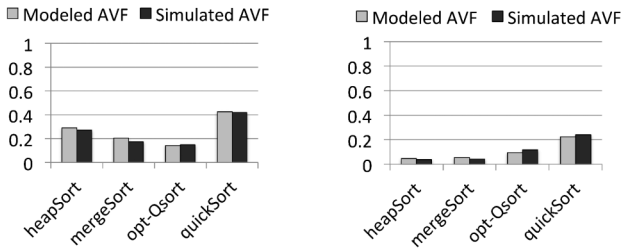
(c) Characteristics of the compiled binaries

Fig. 11. Effect of compiler optimizations on AVF induced by the quick sort algorithm on the *wide-issue machine*.

Despite having a large memory footprint with frequent nonoverlapping L2 data cache and TLB misses, the induced ROB AVF of the quick sort algorithm is relatively low. Intuitively, quick sort involves comparing the pivot value with the data, which results in a load-compare-branch dependence chain. As the input data is random, this branch is inherently unpredictable, resulting in low AVF in the shadow of the L2 miss. The average ROB AVF (Figure 11(b)) across the five quick sort binaries is very similar. The dependence length between a nonoverlapped data L2/TLB miss and a mispredicted branch in its shadow is also comparable. As the data is the same, the total number of such nonoverlapped misses is also similar. However, the mechanisms by which this AVF is induced are contrasting: from Figure 11(c), *qsort-gcc-O1* and *qsort-cc-O1* induce periods of very high AVF during ideal execution. On the other hand, the optimized binaries have very low AVF during ideal execution. Owing to a significant reduction in I-count due to aggressive optimization, these binaries also have a greater frequency of data L2/TLB misses per cycle, but a higher proportion of un-ACE bits (approximately 20% lower than the unoptimized case), resulting in similar overall *average* AVF induced in the ROB.

Figure 11(b) also indicates that the highly optimized binaries induce higher AVF in the LQ. From Figure 11(c), it is apparent that the highly optimized binaries have a greater proportion of loads Per Kilo Instructions, resulting in greater occupancy of the LQ. Furthermore, these binaries have high ILP, resulting in lower dispatch-to-issue latency for load instructions (see Section 3.3); these loads spend a smaller fraction of their lifetime in the IQ as compared to the unoptimized case. In the case of our *wide-issue machine*, each LQ entry has more bits than each IQ entry, resulting in higher overall SER for the machine running the optimized workload (assuming both structures have the same intrinsic fault rate).

Figure 11(c) also compares the relative SER observed in the wide-issue machine while running these traces. The SER is computed for SQ, LQ, ROB, IQ, and FU, assuming a constant intrinsic fault rate, and normalized to the SER of the binary *qsort-gcc-O1*. From Figure 11(c), we note that the unoptimized binaries have lower SER as they have lower occupancies in the LQ and SQ, which have the largest number of bits per entry



(a) Impact of Algorithm on ROB AVF

(b) Impact of Algorithm on LQ AVF

	α	β	O_{ROB}^{ideal}	N_{Data} MPKI	N_{ld} PKI	CPI	I-count (relative)	SER (relative)	Exec. time ($cycles$)	CF/GHz
heapSort	1.46	1.34	128	1.15	160.0	0.38	1.0	1.0	0.38	0.38
mergeSort	1.26	1.69	35.7	1.83	157.0	0.43	2.24	0.66	0.96	0.63
opt-Qsort	1.06	2.16	32.19	2.63	310.0	1.02	0.83	0.61	0.85	0.52
quickSort	1.21	2.26	22.74	1.40	351.7	0.71	2.82	2.00	2.00	4.00

(c) Characteristics of the compiled binaries

Fig. 12. Effect of algorithms on AVF.

(see Table I). These binaries instead have more entries in the IQ, which has fewer bits per entry.

The metric CF/GHz proposed earlier allows us to understand the tradeoff between performance and vulnerability to soft errors on the same microarchitecture. The unoptimized binaries have lower SER but much longer runtimes, making them more vulnerable overall, as compared to the highly optimized Alpha `cc` binaries, as indicated by their CF/GHz in Figure 11(c). However, workload *qsort-gcc-O3* has the least value for CF/GHz, indicating a favorable tradeoff between soft error vulnerability and performance; it lies between the unoptimized binaries and the Alpha `cc`-optimized binaries in terms of ILP, CPI, and utilization of the LQ and SQ. A compiler designer interested in compiling for lower vulnerability may use the model to quickly estimate the impact of an optimization on AVF, SER, and overall vulnerability.

5.3.3. Effect of Algorithms on AVF. In this section, we study the impact of altering the algorithm or the implementation of the algorithm on AVF. Figure 12 illustrates the AVF induced in the ROB and the LQ while running *heap sort*, *merge sort*, *optimized quick sort (opt-Qsort)*, and *nonoptimized quick sort (quick sort)*. Figure 12(c) contains data obtained using our profilers. We refer the reader to Section 5.3.2 for an explanation of each column. The (unoptimized) *quick sort* algorithm is identical to the one used in Section 5.3.2. Workloads *heap sort* and *merge sort* are based on the recursive implementations described in Cormen et al. [2001]. The *opt-Qsort* implementation is based on an algorithm described by Bentley and McIlroy [1993]. This implementation mitigates the overhead of recursion, picks a pseudo-median value as the pivot to mitigate pathological cases for quick sort, and switches to insertion sort once the array sizes are sufficiently small. Less recursion also allows the compiler to aggressively optimize the code. We compile these sorting routines with Alpha `cc`, with `-Ofast` optimization flags.

It is clear from Figure 12(a) that *opt-Qsort* induces the least AVF in the ROB, despite having the highest nonoverlapped Data L2/TLB MPKI. This is primarily due to a reduction in the average dependence length between the nonoverlapped data L2/TLB miss and the dependent mispredicted branch in its shadow. On the other hand, *heap sort* has low ILP, resulting in high ROB occupancy during ideal execution, as seen in Figure 12(c). As seen earlier with the case of the `-O1` optimized *qsort* algorithm, the low ILP results in higher dispatch-to-issue latency for loads. Further, the low frequency of

Table V. Impact of Input Data on AVF and SER

	α	β	O_{ideal}^{ROB}	N_{Data} MPKI	N_{Ld} PKI	ROB AVF	LQ AVF	CPI	I-Count (Relative)	SER (Relative)	Exec. Time (Cycles)	CF/GHz
heapSort	1.46	1.34	128	1.18	158.1	0.30	0.04	0.35	1.15	1.11	0.40	0.44
mergeSort	1.34	1.66	34.26	2.04	177.1	0.25	0.08	0.44	1.47	0.95	0.65	0.61
opt-Qsort	1.0	2.15	28.22	4.86	268.5	0.35	0.30	1.58	0.13	1.43	0.21	0.30
quickSort	1.24	2.24	22.79	1.77	371.9	0.54	0.36	0.89	147.5	2.63	131.3	345.3

loads Per Kilo Instructions results in very low LQ AVF. *Optimized quick sort (opt-Qsort)* has a higher proportion of loads Per Kilo Instruction but lower average occupancy of state in the shadow of a data L2/TLB miss, resulting in lower ROB and LQ AVF. The *quick sort* implementation, however, has a high proportion of loads as compared to *merge sort* and *heap sort*, which, combined with the higher ROB occupancy, is reflected in the LQ AVF.

For the random input, *heapsort* has the least execution time, followed by *opt-Qsort*. It is clear that the *heap sort* implementation is also superior to the *opt-Qsort* implementation in terms of overall vulnerability to soft errors (CF/GHz); although the *opt-Qsort* implementation induces the least SER for the structures under consideration, it is not sufficient to compensate for its higher runtime compared to the *heap sort* implementation. *Opt-Qsort* clearly induces less overall vulnerability than either *merge sort* or *quick sort*. Our model provides a method for quickly comparing the AVF induced using various algorithms and characterizing their properties. Based on the insight provided using the model, algorithm designers may alter their implementation such that performance is maximized while simultaneously minimizing SER.

5.3.4. Impact of Input Data on AVF. It is well understood that input data can affect the performance of a workload. In this section, we demonstrate that input data may have a significant bearing on the AVF induced by a workload on a microarchitecture.

As noted in Section 5.3.2, the AVF induced in the ROB by the sorting algorithms were low for the quick sort implementations, as the randomness of the input data caused many of the branches in the shadow of a data L2/TLB miss to be mispredicted. Intuitively, this suggests that if the data were such that the branches were biased, the AVF induced would be much larger. An already sorted input would bias the comparison operations between two data points, which would bias the branches. Depending on the algorithm and implementation, this may introduce a significant change in the AVF induced while running the workload.

Table V indicates the AVF and SER induced when sorting is applied on a presorted array of the same 500,000 coordinates used in the earlier studies. The I-count and SER results presented are computed relative to the *heapsort* results in Figure 12(c). A sorted input has no significant impact on the working of *heap sort*, as the algorithm requires the creation of a heap; *heap sort* always has a time complexity of $O(n \cdot lg(n))$. In the case of *merge sort*, the partitioning phase remains unchanged, but the merge operation is simplified, resulting in a time complexity of $O(n)$. Applying *quick sort* to a sorted input triggers the worst-case behavior in the workload, making its time complexity $O(n^2)$. The optimized version *opt-Qsort* avoids this through the use of the pseudo-median pivot selection strategy. Furthermore, it switches to insertion sort once the array size is relatively small, which has a time complexity of $O(n)$ for sorted inputs. The change in the input increases the induced SER for all sorting algorithms, as it results in a decrease in the number of branch mispredictions. Consequently, there is an increased proportion of correct-path state in the shadow of a long-latency L2 miss, resulting in higher SER. The extent to which this occurs varies with the algorithm. It is clear from Table V that there is very little change in either the CPI or I-count for *heap sort*. There is a slight increase of 11% in the induced SER due to fewer mispredictions. The CPI

for *merge sort* is nearly identical to the random input case, but there is a 44% increase in SER and a 34% reduction in the I-count. Consequently, the same workload induces higher AVF in the ROB and higher SER overall. The CPI for *opt-Qsort* increases by 54%, the SER increases by 57%, and the I-count reduces by 85%. The CPI for *quick sort* increases by 25%, the SER increases by 31%, and the I-count increases by 5,130%. We see that a simple change in input data may affect CPI, SER, both, or neither, depending on the algorithm and implementation.

Comparing the data presented in Table V and Figure 12, we can explain the underlying mechanisms affecting the AVF and SER of these workloads. For *heap sort*, the values of α, β, ideal occupancy, loads, and nonoverlapped loads per thousand instructions remain nearly identical. Consequently, the ROB and LQ AVF, CPI, I-count, and SER are also largely unchanged. There is a slight increase in AVF that is attributed to fewer dependent mispredicted branches in the shadow of the data L2 or TLB misses. *Merge sort* has a larger number of nonoverlapped data MPKI and a greater proportion of loads PKI, resulting in higher overall SER. The same factors are also responsible for an increase in the AVF and SER for *opt-Qsort* and the unoptimized *quick sort* implementation. Additionally, better branch prediction accuracy leads to a significant increase in the proportion of ACE state in the shadow of a nonoverlapped L2 miss for these workloads.

It is clear that for this input, *opt-Qsort* has the lowest execution time and CF of all workloads, making it the least vulnerable workload. Our *heap sort* implementation has the second-lowest overall vulnerability and the second-highest performance for this input set on the *wide-issue* machine. We thus see that *heap sort* has consistent performance and induced SER for both random and sorted data. If this is a desirable feature for the program being designed or if the data is known to be nearly random in the typical case, *heap sort* would be the best choice on the *wide-issue* machine. As the CF of *heap sort* is 27% lower than *opt-Qsort* for random input data but is 47% higher for sorted data, *opt-Qsort* may be the best choice if the data is expected to be sorted or nearly sorted in the typical case. The CFs of *merge sort* in the random input and the sorted input scenarios are nearly identical. The reduction in execution time is offset by an increase in SER, such that the product of the two remains nearly the same. It is seen that CF may remain identical, change very little (*merge sort, heap sort*), decrease (*opt-Qsort*), or significantly increase (*quick sort*), depending on the input data.

The algorithm designer may thus select the best sorting algorithm based on the knowledge of input data, microarchitecture, and SER. Our data also indicates that if the vulnerability induced by *heap sort* could be reduced through the reduction in CPI, I-count, or SER, it would be superior to the *opt-Qsort* implementation. From the perspective of an algorithm designer/programmer, SER can only be affected by reducing the occupancy of ACE state in vulnerable queues. I-count and CPI can be improved through careful optimization of the code. This optimization may lead to an increase in ACE state in the core structures, as seen in the previous compiler optimization study. This involves careful tradeoffs, which can be quickly evaluated using our model.

5.3.5. Discussion. In the preceding sections, we studied the impact of compiler optimizations, algorithmic changes, and input data on AVF and SER. Our study covers only a subset of experiments necessary to determine the best sorting strategy for a wide range of inputs and input sizes. Cache locality, or lack thereof, at larger input sizes may lead to different conclusions, and the algorithm designer must choose the appropriate algorithm to match the microarchitecture and data. We show that our model enables a quick evaluation a large number of inputs, algorithms, implementations, and optimizations on performance, AVF, and SER. From our studies, it can be seen that in the first order, the choice of algorithm or implementation has a significant impact on

AVF and SER. Compiler optimizations have a significant, albeit less dramatic influence on AVF and SER.

Workload characterization for performance projection typically relies largely on having workloads with a large range of microarchitectural parameters, such as CPI, instruction mix, ILP, MLP, number of last-level cache misses, and so forth. This article demonstrates that AVF and SER are additionally influenced by the interaction between these events and by other events that may have little impact on performance. The characteristics presented in this work may be used to evaluate the coverage of a workload suite for AVF and SER. This work motivates the need for a methodology for AVF benchmarking, orthogonal to any methodology used for performance. A workload suite with a high level of coverage for the various factors identified in this work will help guide correct tradeoffs for SER mitigation and provide confidence in the SER projections. Our methodology is the first that identifies a set of workload characteristics that influence AVF and provides a basis for workload characterization for AVF. These insights thus enable the development of a robust benchmark suite for AVF and SER evaluation.

We also envision our model being used to select optimal algorithms, implementations, or compiler transformations for SER. An implementation of an algorithm can be quickly evaluated for improvement of AVF and SER, and the insight from the model can be used to further optimize it for AVF/SER. Special cases that worsen AVF, such as input data, can be immediately identified and corrected.

6. RELATED WORK

Mukherjee et al. [2003] use Little's law as a high-level technique to estimate occupancy of state in the structure; however, this methodology still requires detailed simulation to extract the IPC and the average latency of each *correct-path* instruction in each structure. Computing the latter from profiling is nontrivial for an out-of-order processor due to overlapping of some execution latencies and dependence on the latencies of other instructions in that structure. Furthermore, it fails to provide insight into the fundamental factors affecting the occupancy of correct-path state beyond aggregate metrics.

As AVF represents the combined effect of the workload and its interaction with the hardware, Sridharan and Kaeli [2009] attempt to decouple the software component of AVF from the hardware component through a microarchitecture-independent metric called *Program Vulnerability Factor* (PVF). PVF has been shown to model the AVF of the Architected Register File using inexpensive profiling. However, for estimating the AVF of other structures, their methodology relies on the estimation of *Hardware Vulnerability Factor* (HVF) [Sridharan and Kaeli 2010], which in turn requires detailed simulation and thus provides less insight than our model. Sridharan and Kaeli have shown that HVF correlates with occupancy of structures such as the ROB, and hence we expect that our modeling methodology can be used to model HVF of the applicable structures.

Fu et al. [2006] report a "fuzzy relationship" between AVF and simple performance metrics such as IPC, cache hit/miss rates, branch mispredict rates, and so forth. As we have observed in earlier discussions, it is the interaction of these events in a CPU that influences AVF and SER. Black-box statistical models for AVF that utilize multiple microarchitectural metrics have been proposed by Walcott et al. [2007] and Duan et al. [2009] for dynamic prediction of AVF. These models use metrics such as average occupancy and cumulative latencies of instructions in various structures as inputs to the statistical model, which are not available without detailed simulation. Although these models do not clearly identify the mechanisms that affect AVF induced in a unit, they indicate that these microarchitectural events do influence AVF. Cho et al. [2007]

utilize a neural-network-based methodology for design space exploration and use it to model AVF of the IQ. As each workload is associated with its own neural network model, training it would potentially require a significant amount of detailed simulations. All these models combine the software and hardware components of AVF and do not uncover the fundamental mechanisms influencing AVF, thereby providing less insight than our approach. As we derive the factors affecting AVF from first principles that explicitly model this fuzzy relationship, we can identify the precise cause of high or low AVF in a particular structure and characterize workloads for AVF.

7. CONCLUSION

In this work, we developed a first-order mechanistic model for AVF, derived from first principles of out-of-order execution, to provide quantifiable insight into the factors affecting the AVF of microarchitectural structures. The modeling methodology requires inexpensive profiling and computes AVF with mean absolute error of less than 7% for the ROB, LQ, SQ, IQ, and FU. Additionally, the model quantifies the impact of each microarchitectural event on AVF and SER. We have demonstrated that the model can be used for understanding how microarchitecture affects AVF. The model can be used to perform design space exploration, to evaluate the impact of parametric changes, and to perform workload characterization for AVF. By modeling the complex relationship between various miss events that affect the occupancy of state in the processor, we are able to quantitatively explain the lack of correlation between AVF and aggregate metrics observed in earlier work. This work enables the architect to identify workloads that would induce high AVF in CPU structures. We presented a case study on using the model to study the impact of compiler optimizations, algorithmic and implementation changes, and input data on AVF, SER, and performance to enable compiler writers, architects, and software designers to make informed choices about minimizing AVF or SER while maximizing performance.

ACKNOWLEDGMENTS

We are grateful to Arijit Biswas (Intel) for his guidance. We thank Faisal Iqbal, Mattan Erez, and all reviewers for their comments and suggestions.

REFERENCES

Robert C. Baumann. 2005. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability* 5, 3 (Sept. 2005), 305–316.

Jon L. Bentley and M. Douglas McIlroy. 1993. Engineering a sort function. *Software- Practice and Experience* 23, 11 (Nov. 1993), 1249–1265.

Shekhar Borkar. 2005. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro* 25, 6 (Nov.-Dec. 2005), 10–16.

Doug Burger and Todd M. Austin. 1997. The simplescalar tool set, version 2.0. *SIGARCH Computer Architecture News* 25, 3 (June 1997), 13–25.

Michael Butler and Yale Patt. 1992. An investigation of the performance of various dynamic scheduling techniques. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO'25)*. 1–9.

Chang-Burm Cho, Wangyuan Zhang, and Tao Li. 2007. Informed microarchitecture design space exploration using workload dynamics. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. 274–285.

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.

Melina Demertzi, Murali Annavaram, and Mary Hall. 2012. Analyzing the effects of compiler optimizations on application reliability. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC'12)*. 184–193.

Lide Duan, Bin Li, and Lu Peng. 2009. Versatile prediction and fast estimation of Architectural Vulnerability Factor from processor performance metrics. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture*. 129–140.

Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. 2009. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems* 27, 2, Article 3 (May 2009), 37 pages.

Stijn Eyerman, James E. Smith, and Lieven Eeckhout. 2006. Characterizing the branch misprediction penalty. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2006*. 48–58.

Xin Fu, J. Poe, Tao Li, and José A. B. Fortes. 2006. Characterizing microarchitecture soft error vulnerability phase behavior. In *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 147–155.

Mohamed A. Gomaa and T. N. Vijaykumar. 2005. Opportunistic transient-fault detection. In *Proceedings of 32nd International Symposium on Computer Architecture*. 172–183.

Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization, 2001 (WWC-4. 2001)*. 3–14.

Tejas S. Karkhanis and James E. Smith. 2004. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture, 2004*. 338–349.

Tejas S. Karkhanis and James E. Smith. 2007. Automated design of application specific superscalar processors: an analytical approach. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. 402–411.

Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, 469–480.

Xiaodong Li, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. 2005. SoftArch: An architecture level tool for modeling and analyzing soft errors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*. 496–505.

Pierre Michaud, Andre Seznec, and Stephan Jourdan. 1999. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*. 2–10.

Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. 29–40.

Arun Arvind Nair, Stijn Eyerman, Lieven Eeckhout, and Lizy Kurian John. 2012. A first-order mechanistic model for architectural vulnerability factor. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. 273–284.

Arun Arvind Nair, Lizy Kurian John, and Lieven Eeckhout. 2010. AVF stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*. 125–136.

Steven K. Reinhardt and Shubhendu S. Mukherjee. 2000. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 25–36.

Edward M. Riseman and Caxton C. Foster. 1972. The inhibition of potential parallelism by conditional jumps. In *IEEE Transactions on Computers*, Vol. 21. Issue 12. IEEE Computer Society, Washington, DC, 1405–1411.

Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. 2007. JouleSort: A balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. ACM, New York, NY, 365–376.

Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. 45–57.

P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the International Conference on Dependable Systems and Networks*. 389–398.

Vilas Sridharan, David Kaeli, and Arijit Biswas. 2007. Reliability in the shadow of long-stall instructions. In *Proceedings of the 3rd Workshop on System Effects of Logic Soft Errors.*

Vilas. Sridharan and David R. Kaeli. 2009. Eliminating microarchitectural dependency from Architectural Vulnerability. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture*. 117–128.

Vilas Sridharan and David R. Kaeli. 2010. Using hardware vulnerability factors to enhance AVF analysis. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. 461–472.

Kristen R. Walcott, Greg Humphreys, and Sudhanva Gurumurthi. 2007. Dynamic prediction of architectural vulnerability from microarchitectural state. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. 516–527.

Nicholas J. Wang, Justin Quek, Todd M. Rafacz, and Sanjay J. Patel. 2004. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*. 61–70.

Christopher Weaver, Joel Emer, Shubhendu S. Mukherjee, and Steven K. Reinhardt. 2004. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*. 264–275.

Received September 2013; revised June 2014; accepted September 2014